

ASG-DesignManager™ User's Guide

Version: 1.4.3

Publication Number: DSR0200-143-UG

Publication Date: December 1983

The information contained herein is the confidential and proprietary information of Allen Systems Group, Inc. Unauthorized use of this information and disclosure to third parties is expressly prohibited. This technical publication may not be reproduced in whole or in part, by any means, without the express written consent of Allen Systems Group, Inc.

© 1998-2001 Allen Systems Group, Inc. All rights reserved.

All names and products contained herein are the trademarks or registered trademarks of their respective holders.



ASG Worldwide Headquarters Naples, Florida USA | asg.com

1333 Third Avenue South, Naples, Florida 34102 USA Tel: 941.435.2200 Fax: 941.263.3692 Toll Free: 1.800.932.5536

ASG Support Numbers

ASG provides support throughout the world to resolve questions or problems regarding installation, operation, or use of our products. We provide all levels of support during normal business hours and emergency support during non-business hours. To expedite response time, please follow these procedures.

Please have this information ready:

- Product name, version number, and release number
- List of any fixes currently applied
- Any alphanumeric error codes or messages written precisely or displayed
- A description of the specific steps that immediately preceded the problem
- The severity code (ASG Support uses an escalated severity system to prioritize service to our clients. The severity codes and their meanings are listed below.)

If You Receive a Voice Mail Message:

- 1 Follow the instructions to report a production-down or critical problem.
- 2 Leave a detailed message including your name and phone number. A Support representative will be paged and will return your call as soon as possible.
- 3 Please have the information described above ready for when you are contacted by the Support representative.

Severity Codes and Expected Support Response Times

| Severity | Meaning | Expected Support Response Time |
|----------|---|--------------------------------|
| 1 | Production down, critical situation | Within 30 minutes |
| 2 | Major component of product disabled | Within 2 hours |
| 3 | Problem with the product, but customer has work-around solution | Within 4 hours |
| 4 | "How-to" questions and enhancement requests | Within 4 hours |

ASG provides software products that run in a number of third-party vendor environments. Support for all non-ASG products is the responsibility of the respective vendor. In the event a vendor discontinues support for a hardware and/or software product, ASG cannot be held responsible for problems arising from the use of that unsupported version.

Business Hours Support

| Your Location | Phone | Fax | E-mail |
|---------------------------------|---|-----------------|--------------------|
| United States and Canada | 800.354.3578 1.941.435.2201 Secondary Numbers: 800.227.7774 800.525.7775 | 941.263.2883 | support@asg.com |
| Australia | 61.2.9460.0411 | 61.2.9460.0280 | support.au@asg.com |
| England | 44.1727.736305 | 44.1727.812018 | support.uk@asg.com |
| France | 33.141.028590 | 33.141.028589 | support.fr@asg.com |
| Germany | 49.89.45716.300 | 49.89.45716.400 | support.de@asg.com |
| Singapore | 65.224.3080 | 65.224.8516 | support.sg@asg.com |
| All other countries: | 1.941.435.2201 | | support@asg.com |

Non-Business Hours - Emergency Support

| Your Location | Phone | Your Location | Phone |
|---------------------------------|--|----------------------------|--------------------|
| United States and Canada | 800.354.3578 1.941.435.2201 Secondary Numbers: 800.227.7774 800.525.7775 Fax: 941.263.2883 | | |
| Asia | 011.65.224.3080 | Japan/Telecom | 0041.800.9932.5536 |
| Australia | 0011.800.9932.5536 | New Zealand | 00.800.9932.5536 |
| Denmark | 00.800.9932.5536 | South Korea | 001.800.9932.5536 |
| France | 00.800.9932.5536 | Sweden/Telia | 009.800.9932.5536 |
| Germany | 00.800.9932.5536 | Switzerland | 00.800.9932.5536 |
| Hong Kong | 001.800.9932.5536 | Thailand | 001.800.9932.5536 |
| Ireland | 00.800.9932.5536 | United Kingdom | 00.800.9932.5536 |
| Israel/Bezeq | 014.800.9932.5536 | | |
| Japan/IDC | 0061.800.9932.5536 | All other countries | 1.941.435.2201 |

ASG Web Site

Visit <http://www.asg.com>, ASG's World Wide Web site.

Submit all product and documentation suggestions to ASG's product management team at <http://www.asg.com/products/suggestions.asp>

If you do not have access to the web, FAX your suggestions to product management at (941) 263-3692. Please include your name, company, work phone, e-mail ID, and the name of the ASG product you are using. For documentation suggestions include the publication number located on the publication's front cover.

Contents

| | |
|--|------|
| Preface | vii |
| About this Publication | vii |
| Publication Conventions | viii |
| Notation For Statement Formats | ix |
| 1 Introduction to DesignManager | 1 |
| What is DesignManager? | 1 |
| DesignManager and Logical Database Design Models | 1 |
| The DesignManager Approach | 3 |
| Identification of Data Elements | 3 |
| Specification of Userviews | 3 |
| Logical Database Design Model Production | 5 |
| Design Analysis and Refinement | 5 |
| Implementation of the Logical Database Design Model | 6 |
| The DesignManager Modeling Dictionary | 6 |
| Structure of a Modeling Dictionary | 6 |
| Modeling Dictionary Member Types | 7 |
| Modeling Dictionary Records | 8 |
| The Automatic Error Recovery System | 10 |
| The Controller | 11 |
| The Security System | 11 |
| DesignManager Safety and Control Features | 12 |
| Dictionary Mode And Design Mode | 13 |
| 2 DesignManager Language and Coding | 15 |
| Statement Identifiers and Statement Bodies | 16 |
| Terminators | 18 |
| Rules Governing Variables | 19 |
| Amendments | 22 |
| 3 DesignManager Commands | 23 |
| Overview | 23 |
| Controller's Commands | 24 |
| Effects of Optional Additional Facilities | 24 |
| Integrated/Non-integrated Manager Products Installations | 24 |
| Member Names in Commands | 24 |

| | |
|--|-----|
| Mode Command Specifications | 25 |
| MODE | 25 |
| Overview of the Dictionary Mode Commands | 25 |
| Modeling Dictionary Access Security Commands | 26 |
| Modeling Dictionary Manipulation Commands | 26 |
| Modeling Dictionary Documentation Commands | 27 |
| Dictionary Mode Command Specifications | 28 |
| Primary Command Specifications | 28 |
| ADD | 28 |
| AUTHORITY | 31 |
| COPY | 32 |
| DICTIONARY | 33 |
| ENDDSR | 37 |
| ENVIRONMENT | 38 |
| LIST | 38 |
| MODIFY | 44 |
| PRINT | 60 |
| REMOVE | 61 |
| Specifications of Secondary Keywords and Clauses | 62 |
| Name-related-selection Clauses | 63 |
| Time-and-user-related-selection Clauses | 67 |
| Overview of the Design Mode Command | 71 |
| Workbench Design Area Manipulation Commands | 71 |
| Workbench Design Area Documentation Commands | 72 |
| Design Mode Command Specifications | 73 |
| CLEAR | 73 |
| DESIGN | 74 |
| ECHO | 75 |
| ENDDSR | 76 |
| FETCH | 76 |
| LIST | 77 |
| MERGE | 79 |
| NAME | 82 |
| NO-ECHO | 84 |
| REPORT | 84 |
| SNAPSHOT | 86 |
| STORE | 87 |
| | |
| 4 DesignManager Member Definition Statements | 89 |
| Overview of the Member Definition Statements | 89 |
| Data Definition Statement Specifications | 91 |
| ITEM Keyword | 91 |
| GROUP Keyword | 92 |
| USERVIEW Keyword | 94 |
| VIEWSET Keyword | 97 |
| Common Clause Specifications | 99 |
| ACCESS-AUTHORITY Clause | 99 |
| ADMINISTRATIVE-DATA Clause | 100 |
| ALIAS Clause | 101 |
| CATALOGUE Clause | 101 |

| | | |
|---|---|-----|
| | COMMENT Clause | 102 |
| | DESCRIPTION Clause | 103 |
| | EFFECTIVE-DATE Clause | 103 |
| | FREQUENCY Clause | 104 |
| | NOTE Clause | 105 |
| | OBSOLETE-DATE Clause | 105 |
| | QUERY Clause | 106 |
| | SECURITY-CLASSIFICATION Clause | 107 |
| | SEE Clause | 108 |
| 5 | DesignManager Output | 109 |
| | Messages | 110 |
| | Echoed Input Lines | 111 |
| | Dictionary Mode Output | 112 |
| | Design Mode Output | 112 |
| | Design Reports | 113 |
| 6 | DesignManager Procedures and Use | 123 |
| | Formulation of Userviews | 123 |
| | Structure and Definitions | 124 |
| | Userview Development | 125 |
| | Naming the Data Elements | 125 |
| | Top-down Approach to Userview Formulation | 130 |
| | Merging Userviews into the Workbench Design Area | 130 |
| | Generating a Third Normal Form Relational Schema | 132 |
| | Relations and Keys | 134 |
| | An Example Illustrating Normalization Benefits | 137 |
| | The DesignManager Design Procedure | 138 |
| | Naming Relations and Storing Them in the Dictionary | 144 |
| | Use of Pseudo-FDs to Handle Subcategories | 151 |
| 7 | Optional Additional Facilities | 165 |
| | User Formatted Output (DSR-UD30) | 166 |
| | Enterprise Modeling (DSR-EM10) | 166 |
| | Load Factor Calculation (DSR-PH10) | 167 |
| 8 | DesignManager/DataManager Integration | 169 |
| | Effects on DesignManager Dictionary Mode Commands | 170 |
| | Effects on DesignManager Design Mode Commands | 170 |
| | The DesignManager MERGE Command | 170 |
| | DataManager Commands Available In Design Mode | 171 |
| | Effects on DataManager Commands | 172 |
| | Effects on Dictionary Member Types | 175 |
| | Effects on DesignManager Optional Additional Facilities | 175 |
| | User Formatted Output (DSR-UD30) | 176 |
| | User Printer Graphics (DSR-UD31) | 176 |
| | Effects on DataManager Optional Additional Facilities | 177 |
| | Process Member Types | 177 |
| | User Defined Syntax (UDS) | 177 |
| | Effects on Language and Coding | 178 |

| | | |
|----|--|-----|
| 9 | Interactive Front-end Facility | 179 |
| | Overview of the Interactive Front-end | 179 |
| | Using the ESCAPE Character | 180 |
| | The Display Areas | 181 |
| | The EXECUTE Command | 182 |
| | Overview of Interactive Front-end Commands | 182 |
| | The :SET Command | 183 |
| | Commands to Examine the Display Areas | 183 |
| | Commands to Edit the Save Area | 184 |
| | Macro Usage Commands | 186 |
| | Error Messages | 186 |
| | Command Specifications | 187 |
| | & (Ampersand) | 187 |
| | :+ (Plus sign) | 188 |
| | :- (Minus sign) | 188 |
| | :ADD | 189 |
| | :BOTTOM | 190 |
| | :CHANGE | 190 |
| | :COPY | 192 |
| | :DELETE | 193 |
| | :DOWN | 194 |
| | :DUPLICATE | 194 |
| | :FAIL | 195 |
| | :INSERT | 196 |
| | :LOCATE | 197 |
| | :MACRO | 198 |
| | :MEND | 199 |
| | :MOVE | 199 |
| | :NUMBERS | 200 |
| | :POSITION | 201 |
| | :PROMPT | 201 |
| | :READ | 201 |
| | :SCCHANGE | 202 |
| | :SET | 204 |
| | :SHIFT | 204 |
| | :SKIP | 205 |
| | :SWITCH | 206 |
| | :TOP | 206 |
| | :VERIFY | 207 |
| | :UP | 207 |
| | :WIPE | 207 |
| | :WRITE | 208 |
| | :EXECUTE | 208 |
| 10 | User Printer Graphics (DSR-UD31) | 209 |
| | Introduction | 209 |
| | Command Specifications | 209 |
| | PLOT | 209 |
| | Logical Schema Cluster Plot | 210 |
| | The Individual Logical Schema Cluster | 210 |

| | |
|--|-----|
| The Logical Schema Association Matrix | 212 |
| Network Cluster Plot | 214 |
| Creation of Assumed Relations | 215 |
| Individual Network Clusters | 215 |
| The Network Cluster Association Matrices | 217 |
| Appendix | |
| Technical Background | 221 |
| The Relational Model of a Database | 223 |
| Relations and Dependencies | 223 |
| Rules of Inference for Deriving Additional FDs | 225 |
| Keys and Representation of FDs in a Schema | 227 |
| The Normal Forms Defined | 230 |
| The Extended Synthesis Procedure for Schema Design | 238 |
| DSR-MVD Versus Fagin-MVD | 243 |
| Glossary | 245 |
| Bibliography | 257 |
| Index | 259 |

Preface

This *ASG-DesignManager User's Guide* provides an introduction to the concept of the modeling dictionary and logical database design procedures using ASG-DesignManager (herein called DesignManager), and to the optional additional facilities (selectable units) that are available within DesignManager. This publication also describes the effects of the integration of DesignManager with ASG's data dictionary product DataManager. DesignManager is an information modeling and logical database design system.

Allen Systems Group, Inc. (ASG) provides professional support to resolve any questions or concerns regarding the installation or use of any ASG product. Telephone technical support is available around the world, 24 hours a day, 7 days a week.

ASG welcomes your comments, as a preferred or prospective customer, on this publication or on any ASG product.

About this Publication

This publication consists of these chapters:

- [Chapter 1, "Introduction to DesignManager."](#) provides a general introduction to DesignManager, to the modeling dictionary, and to logical database design using DesignManager.
- [Chapter 2, "DesignManager Language and Coding."](#) describes the design of the DesignManager language and states the rules applying to input statements.
- [Chapter 3, "DesignManager Commands."](#) defines the DesignManager commands with summaries and detailed specifications.
- [Chapter 4, "DesignManager Member Definition Statements."](#) defines the DesignManager commands with summaries and detailed specifications in respect of the input to the modeling dictionary.
- [Chapter 5, "DesignManager Output."](#) describes, in general, the DesignManager output.
- [Chapter 6, "DesignManager Procedures and Use."](#) details the procedures and usage of logical database design using DesignManager.
- [Chapter 7, "Optional Additional Facilities."](#) summarizes the optional additional facilities available with DesignManager and states where they are documented.
- [Chapter 8, "DesignManager/DataManager Integration."](#) gives details of the effects of integrating DesignManager with DataManager.

- [Chapter 9, "Interactive Front-end Facility,"](#) describes the DesignManager optional additional facility Interactive Front-end, and gives summaries and details of the capabilities and commands provided by this facility.
- [Chapter 10, "User Printer Graphics \(DSR-UD31\),"](#) describes the DesignManager optional additional facility User Printer Graphics, and gives summaries and details of the capabilities and commands provided by this facility.

Publication Conventions

Allen Systems Group, Inc. uses these conventions in technical publications:

| Convention | Represents |
|---|--|
| ALL CAPITALS | Directory, path, file, dataset, member, database, program, command, and parameter names. |
| Initial Capitals on Each Word | Window, field, field group, check box, button, panel (or screen), option names, and names of keys. A plus sign (+) is inserted for key combinations (e.g., Alt+Tab). |
| <i>lowercase italic monospace</i> | Information that you provide according to your particular situation. For example, you would replace <i>filename</i> with the actual name of the file. |
| Monospace | Characters you must type exactly as they are shown. Code, JCL, file listings, or command/statement syntax. Also used for denoting brief examples in a paragraph. |
| Vertical Separator Bar () with underline | Options available with the default value underlined (e.g., Y N). |

Notation For Statement Formats

In all publications relating to DesignManager, the following notation is used in the specification of statement formats (for commands and data definition statements):

- All words printed in capitals are statement identifiers or keywords that must be present in full or truncated form in the circumstances stated in the statement specification. The extent beyond which a word must not be truncated is indicated by under-lining of the characters that must be retained.
- All words printed in lower case are variables for which the user must substitute a value consistent with the specification.
- Material enclosed in square brackets [] is an option which may be included or omitted as required.
- Braces $\{ \}$ indicate that a choice must be made of one of the options enclosed within them.
- Three full stops . . . indicate that the material they immediately follow may be repeated. Where . . . immediately follows a closing square bracket or brace, the material that can be repeated is bounded by that square bracket or brace and the corresponding opening square bracket or brace. If material can be repeated only a limited number of times, the repetition permitted is stated in the specification.
- Other punctuation marks and symbols must be coded as shown, subject to the implications of any square brackets or braces enclosing them; except that where a single quote ' , is shown, a double quote, " , can alternatively be used, provided that the opening and closing quotes of any pairs of quotes are the same character (single quote or double quote).

1

Introduction to DesignManager

What is DesignManager?

DesignManager is a dictionary-driven interactive or batch software tool for automatically producing logical database design models from database end users' requirements.

Design requirements are stored in a controlled way in the modeling dictionary and can be input at any time to the DesignManager design procedure.

The design models produced by DesignManager are directly applicable to relational databases and can be converted in a straightforward manner for implementation in other database structures. Throughout this publication, the term database is used in its general sense to include sets of related conventional files.

The modeling dictionary is discussed in ["The DesignManager Modeling Dictionary" on page 6](#) and the logical design model and the DesignManager design facilities in ["The DesignManager Approach" on page 3](#).

DesignManager and Logical Database Design Models

A logical database design model may be regarded as a description of a physical database at a conceptual level. It does not describe directly either the data values that will be present in the database or the physical arrangement of the database.

The need for some form of conceptual description of a database to be defined before the database is structured and implemented has become apparent with the growing use of databases and their associated database management systems. For optimum efficiency of the database, in terms of its performance, ability to be extended to meet new uses, and minimal maintenance costs, the structure of the database must meet certain requirements above the general requirements of the database management system employed. In particular, it is important that the data values held in the database are duplicated as little as possible while meeting all database end users' requirements. Minimal duplication of data values in the database reduces updating problems and costs.

From end users' requirements of the database, DesignManager produces a conceptual description of how the database should be structured. This conceptual description is the logical database design model. In producing the model, DesignManager employs an accepted data analysis technique (called normalization) which ensures that the resulting logical database design model, when implemented in a physical database, will incur values while still meeting all specified end users' requirements.

The value of the logical database design model depends on several factors:

- Whether it describes all the types of data (data elements) that are required in the database.
- Whether it accommodates all the end users' requirements of the database (the userviews).
- Whether, when implemented in the physical database, it minimizes the duplication of the data values that are included in the database.
- How well it can accommodate new database end users' requirements of the database and the addition of further types of data, as the database and the applications that it serves evolve.

Identifying the data elements (the types of data) that are represented in the database is mainly a clerical task which together with identifying the userviews (the database end users' requirements of the database) may be linked to one of many top-down data analysis methodologies; alternatively, it may come from analysis of data flows, existing reports, existing files, interviews, or other company specific approaches.

Whatever approach to this definition stage is adopted by a particular company, DesignManager provides full storage and documentation with cross-referencing facilities in the modeling dictionary.

If it were done manually, the amalgamation, analysis, and structuring of all the data elements and userviews to produce a logical database design model which meets the above requirements for minimal duplication of data values in the database and optimal stability with database evolution would be a time-consuming, iterative, and complex task. Using DesignManager, this process is automatic. The database designers are therefore free to devote themselves to aspects of the design process that require human judgement, such as the evaluation of the logical model and userviews for omissions and errors, the physical implementation of the model in the database, and balancing the advantages and disadvantages of altering the logical model to match the available hardware and meet particular performance requirements.

The logical database design model produced by DesignManager is called, in database terminology, a *third normal form relational schema*. Third normal form describes a structure which has certain inherent properties concerning the duplication of data elements in the design and the flexibility of the database derived from the model to evolution and growth.

A *relational schema* is an overall logical structure which is composed of individual relations. A relation is a description of the arrangement of data values in the database for which the logical database design model has been generated. If the database in which the logical design model is to be implemented is a relational database, these relations correspond with the relations in that database; if the database is not structured by relations but by segments or other entities, the relations, because they describe arrangements of data at a conceptual level, can be modified for implementation in a particular database system.

The DesignManager Approach

These are the steps involved in producing and implementing a logical database design model:

1. Identifying the types of data required in the database—the data elements.
2. Identifying and defining database end users' requirements of the database—the userviews.
3. Amalgamation, analysis, and simplification of the userviews to produce a logical database design model which will incur minimal duplication of data values in the database and will be flexible to future requirements as the database applications evolve.
4. Review of the generated logical database design model for completeness and if necessary a redesign to accommodate omitted userviews.
5. Implementation of the logical design model in a physical database running under a particular database management system.

DesignManager addresses the first three of these steps and facilitates any redesign required.

The structure and facilities of the modeling dictionary as the driving force behind DesignManager are discussed in ["The DesignManager Modeling Dictionary" on page 6](#). The remainder of this section discusses in general terms the use and actions of DesignManager; a more detailed discussion is given in [Chapter 6, "DesignManager Procedures and Use," on page 123](#).

Identification of Data Elements

One of the first stages in the production of a logical database design model is the identification of all the types of data that will be represented in the database. In DesignManager, these types of data are called data elements and are assigned names. For example, if these data values were required in the database they could be represented by a data element called *name*:

F. Harris
A. Brown
J. Smith
B. Jones

In DesignManager, all the data elements required in the database and thus in the logical database design model are stored with descriptive information in the modeling dictionary.

Specification of Userviews

Once the data elements required in the database have been defined in the modeling dictionary, each end user of the database must specify the way in which they wish to access the data values represented by those data elements. In DesignManager, the requirements of each database end user are together called a userview; each requirement within a userview is specified as a dependency between one or more data elements.

A dependency may be viewed as a specification of how a user wishes to access the data values in the database. For example, if a database end user wanted to access a data value represented by the data element called EMPLOYEE-NAME by specifying a data value represented by the data element called EMPLOYEE-NUMBER, they would specify a dependency between the two data elements thus:

EMPLOYEE-NAME is dependent on EMPLOYEE-NUMBER

or

EMPLOYEE-NUMBER determines EMPLOYEE-NAME

This dependency may be written in a short-hand notation:

EMPLOYEE-NUMBER ----> EMPLOYEE-NAME

This is an example of a *functional* or *single-valued* dependency because each data value associated with the data element on the left-hand side of the arrow (the left-hand side of the dependency) uniquely identifies a single data value associated with the data element on the right-hand side of the arrow (the right-hand side of the dependency).

Another type of dependency is a *multivalued* dependency. In a multivalued dependency, each data value associated with the data element on the left-hand side of the dependency identifies zero, one or more data values associated with the data element on the right-hand side of the dependency. For example, if a user wanted to access the data values associated with the data element called EMPLOYEE-CHILD-NAME by specifying a value associated with the data element EMPLOYEE-NAME, they would specify a multivalued dependency between the two data elements. In the short-hand notation this would be written:

EMPLOYEE-NAME ---->> EMPLOYEE-CHILD-NAME

Although the meaning of a dependency is clearer when only one data element occurs on each side, it is possible, and indeed necessary in some cases, for more than one data element to occur on one or both sides. For example, if a functional dependency exists between the data elements EMPLOYEE-NUMBER and EMPLOYEE-NAME, and between the data elements EMPLOYEE-NUMBER and SOCIAL-SECURITY-NUMBER, this could be expressed as:

EMPLOYEE-NUMBER ----> EMPLOYEE-NAME
+ SOCIAL-SECURITY-NUMBER

When two data elements appear on the left-hand side of a dependency, they have a slightly different meaning: that more than one data value (each represented by a different data element on the left-hand side of the dependency) is required to access the data values represented by the data elements on the right-hand side of the dependency. For example, if the data values represented by the data element PRICE were to be accessed by specifying a data value represented by the data element SUPPLIER-NUMBER and a data value represented by the data element PART-NUMBER, this would be written as:

SUPPLIER-NUMBER + PART-NUMBER ----> PRICE

For a complete definition of a database end user's requirements of the database, it is necessary to associate other information with the dependencies. This information consists of estimates of the *relative frequency* with which the user requires access to the data values represented by the data elements in the userview and the *response time* they require for this access. In addition, where multivalued dependencies are specified, the userview should contain an estimate of the number of data values represented by a right-hand side data element which will be accessed when a data value represented by a left-hand side data element is specified. This estimate is called the *multiplicity* of the dependency.

These estimates of relative frequency, response time, and multiplicity are important factors to consider when implementing a logical database design model in a physical database.

Besides the specifications of users' access requirements, dependencies may also be used to store logical associations between data elements (see ["Structure and Definitions" on page 124](#)).

Logical Database Design Model Production

Once all the userviews and data elements that together describe the requirements for the database are defined and stored in the modeling dictionary, they must be amalgamated, analyzed, simplified, and a logical database design model formed.

In DesignManager, this is a three-stage process:

- Amalgamation and preliminary simplification
- Further simplification
- Formation of relations

Dependencies defined in userviews are simplified in two ways. In some instances, extraneous data elements are removed from the dependencies; in other instances, whole dependencies may be removed where the access requirements they describe are fully accommodated by other dependencies defined in other userviews. (See [Chapter 6, "DesignManager Procedures and Use," on page 123](#) for a full description of the procedures involved.)

DesignManager also calculates the expected performance of the logical database design model from the frequency, response time, and multiplicity data associated with each userview.

Design Analysis and Refinement

The logical database design model produced by DesignManager can be reported with the DesignManager reporting facilities (see ["Workbench Design Area Documentation Commands" on page 72](#) and [Chapter 10, "User Printer Graphics \(DSR-UD31\)," on page 209](#)). These reports can be used within an installation as management reports, as a basis for database end users to evaluate the model for completeness and to pin-point any omissions, and for the database analyst to plan the structure of the physical database. If it is found that the userviews input to the design procedure were incomplete or inaccurate, the userviews may be amended and the design procedure repeated. Since all the userviews input to the design procedure are stored in the modeling dictionary, it is a relatively simple task to alter the userviews as necessary and start another design iteration.

Implementation of the Logical Database Design Model

Once an approved logical database design is produced, the database design analyst adapts the model to the particular database structure required at the individual installation. The logical database design model produced by DesignManager is directly applicable to relational databases, or it may be adapted to most popular structures whether hierarchical, network, or inverted file.

The DesignManager Modeling Dictionary

The modeling dictionary is an organized collection of definitions of data relevant to the generation of a logical database design model.

Just as an ordinary English dictionary contains definitions of English words, so the modeling dictionary contains definitions of the types of data (the data elements) and end users' access requirements of this data (in terms of dependencies) that the logical database design model will reflect.

Note: _____

The modeling dictionary does not contain the real data that forms the database resource of an organization, but contains data about that data: its forms, characteristics, and interrelationships. _____

The definitions stored in the modeling dictionary are the main input to the design procedure that produces the logical database design model. Thus design runs can be repeated without unnecessary repetition of input.

Structure of a Modeling Dictionary

A DesignManager modeling dictionary comprises these datasets (files):

- The source dataset
- The data entries dataset
- The index dataset
- The error recovery dataset
- If logging is applied to the modeling dictionary, a log dataset

Data definitions are first written in DesignManager source language, in data definition statements, which are then entered into the source dataset by DesignManager commands. (The DesignManager source language, which includes both data definitions statements and commands, is defined in [Chapter 2, "DesignManager Language and Coding," on page 15](#), [Chapter 3, "DesignManager Commands," on page 23](#), and [Chapter 4, "DesignManager Member Definition Statements," on page 89](#), for basic version of the language; extensions of the language to provide for additional DesignManager facilities are defined in the separate facility publications.)

The source dataset contains the data definitions as input or as subsequently amended (except that consecutive spaces are compressed in order to save storage; but as they are expanded to the input format on output, this is not apparent to the user.)

The data entries dataset contains encoded data definitions generated by DesignManager from the source data definitions either when the source is added to the modeling dictionary or when it is modified. The internal encoded form of the data definitions is required to facilitate fast processing of the information contained in the modeling dictionary. Included with the encoded form of the data definitions is further information, added by DesignManager, regarding the interrelationships between data definitions: see further on this point in ["Modeling Dictionary Records" on page 8](#). If encoded data definitions include aliases or catalogue classifications (see ["Common Clause Specifications" on page 99](#)), DesignManager also generates separate records for these aliases and classifications, and places them in the data entries dataset.

The index dataset contains the name (in IBM internal code hexadecimal sequences) and the address of each data definition that is contained in the source dataset and/or the data entries dataset, and of each alias record and classification record that is contained in the data entries dataset. The index is designed to achieve the fastest possible retrieval of source or encoded data definitions, aliases, and classifications. Also contained in the index dataset is certain additional information obtained from the data definitions, again to facilitate faster processing of some of the DesignManager commands.

The error recovery dataset is used as a temporary back-up file by the automatic error recovery system. The automatic error recovery system is described in ["The Automatic Error Recovery System" on page 10](#).

If logging is applied to a modeling dictionary, a log dataset is established as a fifth dataset of that dictionary. All updating commands, together with any associated data definitions or amendments, are then logged in that dataset as they occur, together with full date, time, user, and physical input/output accesses information.

Modeling Dictionary Member Types

Any data definition for which a record exists in a modeling dictionary is termed a member of that dictionary. For optimum use of the modeling dictionary as a store for all the data definitions relevant to the logical database design, information regarding the nature and interrelationships of that data is necessary. Accordingly, the DesignManager modeling dictionary provides facilities for associating textual information and cross-references to other members with each member. In addition, the modeling dictionary member types have a limited hierarchical arrangement, so that where appropriate, members may be grouped within other members.

These are the basic member types:

- ITEM
- GROUP
- USERVIEW
- VIEWSET

ITEM is a fundamental element of data, the smallest named unit into which data is divided in the particular user organization. An ITEM is hierarchically below the GROUP member type. In DesignManager, the ITEM member type is used to define the data elements or the logical design.

GROUP is a combination of ITEMS and/or other GROUPs; it is thus hierarchically above the ITEM member type. GROUPs can be nested to any depth. In DesignManager, the GROUP member type is used to group a set of data elements, defined individually as ITEM or GROUP members, which is to be treated in the design procedure as a single data element. The GROUP member type can also be used to group data elements prior to the definition of a userview (see ["Top-down Approach to Userview Formulation" on page 130](#)).

USERVIEW is used to store a database end user's view of the database, in terms of dependencies between data elements defined as items or groups. USERVIEWs are the primary input to the DesignManager design procedure. The USERVIEW member type is hierarchically below the VIEWSET member type.

VIEWSET is a combination of USERVIEWs and/or other VIEWSETs. VIEWSETs can be nested to any depth; they provide a convenient method of grouping USERVIEWs prior to their input to the design procedure. VIEWSET member types are hierarchically above the USERVIEW member type.

A member's type is always stated in its data definition statement (see [Chapter 2, "DesignManager Language and Coding," on page 15](#) and [Chapter 4, "DesignManager Member Definition Statements," on page 89](#)).

Some DesignManager optional additional facilities provide for other member types to be defined in the modeling dictionary. These optional additional member types are defined in the facility publication for which they are available.

Modeling Dictionary Records

A modeling dictionary member can have either two or three records in the modeling dictionary. When fully entered into the modeling dictionary, a member has three records; one in each of the source, the data entries, and the index datasets; but at earlier stages a member can exist with only two records. In all cases, one of these records is an index record; the other record can be a source record or a data entries record, depending on how the member is added to the modeling dictionary.

The descriptions which follow are conceptual; the actual techniques used are not described.

When a data definition is first inserted into a modeling dictionary by the ADD command (see [Chapter 3, "DesignManager Commands," on page 23](#)), the command includes the name of the member to insert. If that name is not already recorded in the index dataset, an index record is created for it, and the data definition statement is written into a record in the source dataset.

If the source record is syntactically correct, then as part of the action of the command, an encoded record is generated from the source record and is inserted into the data entries dataset. If the source record is not syntactically correct, no encoded record generates. The source record can be corrected by another command which generates an encoded record and inserts it into the data entries dataset.

The data entries record contains not only the encoded form of the data definition, but also the information generated by DesignManager regarding the member's interrelationships with other members. This information consists at this stage of a set of pointers, pointing to all those members of the dictionary to which reference is made in this member's data definition. These are the *refers to* pointers, or *uses* pointers. For example, if a member named GROUPA is defined as a group containing two other members, GROUPB and GROUPC, then GROUPA's data entries record contains pointers to the data entries records of GROUPB and GROUPC.

There are two possibilities to consider:

- A member to which reference is made may already have a data entries record.
- A member to which reference is made may not have an existing data entries record.

Assume that GROUPB in the example has a data entries record. The information contained in that record regarding GROUPB's interrelationships with other members is now no longer complete. It is necessary to record not only what members are used by a member, but also by what members the member is itself used. Accordingly, GROUPB's data entries record must now have further pointers added, pointing back to GROUPA. These are *referenced by* or *used by* pointers. The insertion of these pointers into GROUPB's data entries record is part of the action resulting from the command which encodes GROUPA.

Thus, the information concerning interrelationships which is inserted into members' encoded records by DesignManager consists of two sets of pointers: the *refers to* or *uses* and the *referenced by* or *used by* pointers.

Now assume that GROUPC has no existing data entries record. In this case, when GROUPA is encoded, DesignManager sets up a dummy data entries record for GROUPC; the dummy record contains GROUPC's *used by* pointers. Later, when the data definition of GROUPC is encoded, DesignManager replaces the dummy record by the encoded record, having first copied the *used by* pointers from the dummy into the encoded record.

GROUPC may or may not already have a source record (and hence an index record) in the modeling dictionary. If it has no source record, an index record for GROUPC is created when the dummy data entries record is set up.

In summary:

- A member can have an index record, a source record, and a data entries record.
- A data entries record can be an encoded record or a dummy record.
- An encoded record can contain an encoded data definition, *refers to* or *uses* pointers, and *referenced by* or *used by* pointers.
- A dummy record can contain *Referenced by* or *used by* pointers only.

The Automatic Error Recovery System

DesignManager incorporates a fully automatic error recovery system. The system is designed to safeguard the modeling dictionary against corruption or loss of data arising from machine malfunction, operator action, or software failure. This section describes how it operates.

When DesignManager receives any command that updates the modeling dictionary, an indicator is set in the dictionary's error recovery dataset. This indicator is called the update lock. As the command processes, any modeling dictionary records that are to be updated are first copied in their pre-update state into the dictionary's error recovery dataset. When processing of the command completes, the update lock is set off and the records are cleared from the error recovery dataset. (The clearance is effected by writing an end of file marker into the first record).

If the DesignManager run is interrupted before the processing of the updating command completes, the update lock remains set on. This triggers the error recovery system on the next occasion that DesignManager processes that dictionary in update mode: either when a subsequent DesignManager job is submitted or when a command is received from a concurrent user. The records in the error recovery dataset are copied back into the dictionary's other datasets, the update lock is set off, and the error recovery dataset clears. This restores the modeling dictionary to the state it was in immediately before receiving the interrupted updating command.

At the end of the recovery process, a message stating that automatic recovery has occurred is output, and DesignManager proceeds to execute the currently entered command.

Because the error recovery dataset is not opened when a dictionary is opened in read-only mode, automatic error recovery cannot take place in read-only mode.

The updating commands are listed below. Their specifications are found in [Chapter 3, "DesignManager Commands," on page 23](#). The REMOVE command can update more than one member. In this case, the error recovery system operates as though a separate command were issued for each member processed; that is, the update lock is set and unset, and the error recovery dataset records are written and cleared, as each individual member is processed. After any recovery action following the interruption of a REMOVE command, therefore, it is not necessary to process again those members whose processing is already complete. These are the updating commands:

- ADD
- REMOVE
- COPY
- MODIFY

Certain Controller's commands are also updating commands. These are indicated in the *ASG-Manager Products Controller's Manual*.

The modeling dictionary's error recovery dataset must be present whenever the dictionary is processed in update mode. It need not be present if the dictionary is open in read-only mode.

The Controller

DesignManager's primary purpose is as a tool for database design analysts to generate automatically logical database designs from information held in the modeling dictionary. Although DesignManager is designed so that end users of the database may be fully involved with the specifications of the requirements of the database, it is likely that the control of the use of DesignManager will be the responsibility of the database design department, and the chief database design analyst in particular.

In DesignManager installations, the Controller has two main functions: to coordinate the approach to the collection of information on which the logical database design model will be based; and to control the security of the DesignManager software and the maintenance of the modeling dictionary datasets. The first function will probably be linked to a chosen top-down database design methodology. The second function is more the concern of this section.

No attempt is made to lay down a job description for the Controller, but certain tasks must be kept within the Controller's personal province if the DesignManager software is to be used with full success. These are the tasks:

- The creation of the modeling dictionary
- The reorganization of the modeling dictionary when necessary
- Securing the modeling dictionary against corruption or loss by maintaining back-up copies of the modeling dictionary
- Securing the modeling dictionary against unauthorized access
- The transfer of information between modeling dictionaries if more than one dictionary is maintained

The Security System

The full specifications of any command mentioned by name in this description of the security system are to be found in [Chapter 3, "DesignManager Commands," on page 23](#). Actions mentioned as performed by the Controller involve private commands that are not available to other users.

The name of each user authorized to access the modeling dictionary is registered in the modeling dictionary by the Controller. For each user, the Controller registers an individual password.

When a modeling dictionary is to be opened for processing in a DesignManager run, it is identified and opened by means of a DICTONARY command. No commands for processing the opened dictionary are accepted by DesignManager until an AUTHORITY command, quoting a password that has been registered in that dictionary as identifying an authorized user, is received. Once a valid AUTHORITY command is received, then subsequent commands are accepted as emanating from the identified user until a further AUTHORITY command or a further DICTONARY command is received.

It is important that users should adhere to any procedures established by the Controller for safe-guarding the secrecy of passwords.

DesignManager Safety and Control Features

DesignManager is designed so that database end users can be fully involved in the specification of their requirements of the database. Certain safety and control features are therefore included in the basic DesignManager software to assist end users in defining their requirements and to help to ensure that meaningful designs are generated. Some of these features relate to the modeling dictionary and some to the design procedure.

Certain syntactical rules must be followed whenever definitions are entered into the modeling dictionary and if any errors are detected a full definition of the member is not added. This helps to ensure the validity of the definitions stored in the modeling dictionary. Because it is important that no incorrect information is included in a logical database design model, DesignManager prevents any userview that is not correctly entered into the modeling dictionary from being input to the design procedure.

It is not necessary for each data element specified in a userview to have been previously defined in the modeling dictionary. As explained in ["Modeling Dictionary Records" on page 8](#), DesignManager creates dummy members for these and reports their creation. The creation of a dummy, however, should be heeded as a warning of one of three things:

- That a spelling mistake has been made in specifying the data element to which the user wished to refer
- That the user is unaware of the name by which the data element is known within the company for the purposes of the logical database design
- That the data element to which the user wished to refer has not been required up to that point in time by any other user

The first two points, if not dealt with carefully, could have serious repercussions on any logical database design generated since they both result in the generation of a synonym, that is, the same data element known by two different names. If synonyms are included in a logical database design, then for each different name by which the same data element is known, a separate data element is included with resulting duplication of data values when the logical design is implemented in a physical database. For this reason, it is suggested that whenever a dummy member is created and reported, the user should search the modeling dictionary thoroughly for any data element already defined which has the same meaning as the one to which they wish to refer, and then correct the userview accordingly.

In the situation where no previously defined data element with the correct meaning can be found, the data element is probably a new one and a definition for it should be entered in the modeling dictionary so that other users can refer to it. Whenever a new data element is defined, the possibility of creating a synonym should be considered. Particular installations may wish to set up standard procedures to implement whenever dummy data elements are created.

To assist further in the prevention of synonyms in a logical database design model, DesignManager can, at the user's option, check that each data element in each userview input to the design procedure has a full definition in the modeling dictionary and is not a dummy.

When many end users throughout a company each specify their own requirements of the database, it is likely that many dependencies are defined in more than one userview. DesignManager recognizes these duplications but represents them only once in the logical database design model.

When two end users specify a dependency between the same data elements, one user may specify the dependency as multivalued and the other as functional. Two such views are inconsistent and may indicate a mistake or misunderstanding on the part of one user. The validity of a logical database design model based on such inconsistencies cannot be guaranteed; DesignManager prevents the generation of such a model, unless otherwise instructed.

The above points are discussed in more detail in [Chapter 6, "DesignManager Procedures and Use," on page 123](#), together with many other aspects on the use of DesignManager.

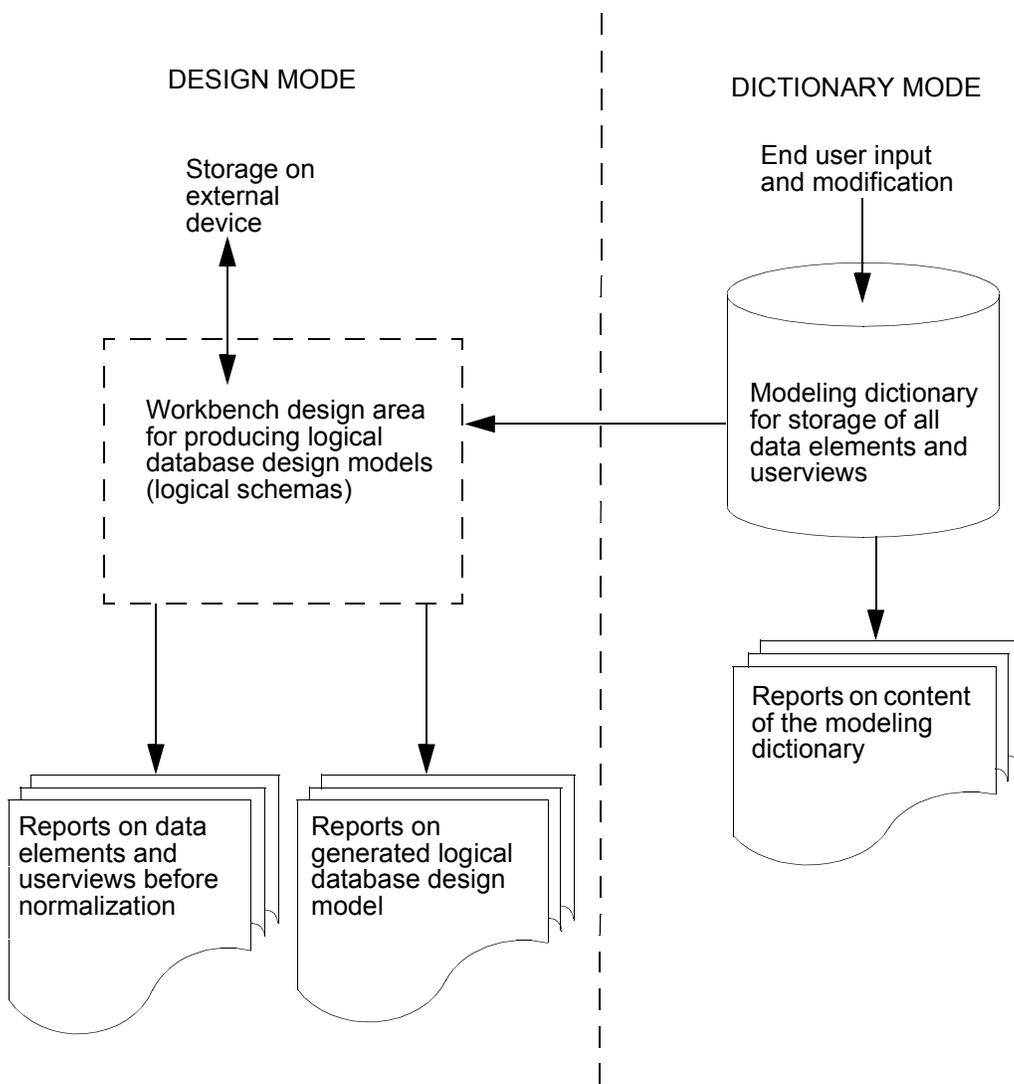
Dictionary Mode And Design Mode

DesignManager is designed to reflect its two main uses: as a storage medium for the data elements and userviews from which the logical database design model will generate; and as a design tool to produce the logical database design model. It therefore works in one of two modes: dictionary mode or design mode. Switching modes involves only one command which may be issued at any time.

The actions that can be performed in one mode are specific to that mode. In dictionary mode, DesignManager usage is confined to adding definitions to the modeling dictionary, and modifying and reporting on that information. In design mode, DesignManager usage is confined to copying information from the modeling dictionary to the workbench design area, manipulating that information to form a logical database design model, and reporting on that information before and after normalization.

Diagrammatically, DesignManager may be viewed as shown in [Figure 1](#).

Figure 1. Diagrammatic overview of DesignManager



2

DesignManager Language and Coding

The DesignManager language consists of three kinds of statements: command statements, member definition statements, and amendments. Within these main divisions, command statements fall into three groups:

- Dictionary mode commands
- Design mode commands
- Mode commands

The statements for commands and for member definitions follow the same coding conventions except where otherwise indicated. Amendments are discussed separately in ["Amendments" on page 22](#). The language is designed to offer the maximum possible flexibility in coding, so that statements are to a high degree free form.

The formats of command statements and the specifications of the commands are given individually for each command in [Chapter 3, "DesignManager Commands," on page 23](#), where they are grouped according to the mode in which they operate. The formats and specifications of the member definition statements are given individually for each member type in [Chapter 4, "DesignManager Member Definition Statements," on page 89](#).

This chapter discusses general points about the structure of DesignManager statements and the rules governing their coding.

All DesignManager statements have at least two parts:

- The statement identifier
- The terminator

In addition, most statements have an additional part, the statement body. Here is the general structure of these statements:

- The statement identifier
- The statement body
- The terminator

Statement Identifiers and Statement Bodies

A statement commences with a statement identifier, which is a keyword. In a command statement, the statement identifier is a command identifier. In a member definition statement, the statement identifier is a member type identifier.

Where applicable, the statement body follows the statement identifier, and can commence on the same line or a separate line.

What constitutes an input line depends on the input device; it can, for instance be a line from a visual display or hard copy terminal, the contents of a punched card, or records from magnetic tape or disk. (In OS environments, records from magnetic tape or disk can be fixed length, variable length, or undefined. In DOS environments, records from magnetic tape or disk must be fixed length unblocked.)

The maximum number of characters acceptable in an input line is 80.

When used *interactively*, statements can continue over any number of input lines, successive lines of input being treated as part of the same statement until a statement terminator is detected.

When used *in batch*, statements issued in design mode can continue over any number of input lines, successive lines of input being treated as part of the same statement until a statement terminator is detected. Statements issued in dictionary mode, including any associated data definition statement or set of amendment lines, must not exceed 409 input lines or the installation defined maximum whichever is less.

When used *interactively*, with the exception of command statements that must be followed by data definition statement or amendment lines, successive command statements may appear entirely or in part on the same input line. That is, the terminator of one command or mode statement may be followed, with or without intervening spaces, by the identifier of another command, the terminator of which may or may not be on the same line.

When used *in batch*, all DesignManager statements must start on a new line.

The statement body consists of a number of keywords and/or variables, with in some cases punctuation symbols (only as required or permitted by the individual statement specifications). Within the statement body, a keyword which controls a subordinate keyword and/or variable is, together with its subordinate keyword and/or variable, called a clause.

Variables can be:

- Names
- Integers
- Decimal numbers
- Floating point numbers
- Character strings
- Undelimited character strings

Rules governing the different types of variables are discussed in ["Rules Governing Variables" on page 19](#).

Individual variables and keywords, including statement identifiers and terminators, are called elements.

Elements must be separated from other elements by one or more spaces, with these exceptions:

- A terminator in a command statement need not (but can) be separated from the last preceding element of the statement.
- In a list of variables, each element must be separated from the next in the list by a comma. The last element in the list is not followed by a comma. A comma can, but need not, be followed and/or preceded by a space or spaces.
- In some commands, where a number of optional keywords are listed, each element in the list can be separated from the next by a comma and/or spaces. Where this applies, it is stated in the command specification. The last element in the list is not followed by a comma.
- If an element ends in the last character position of a line, the next element can (but need not) commence in the first character position of the next line.

Except as stated below, any element can start in any character position of an input line, provided that the element is contained on one line. These are the exceptions:

- The terminator of a member definition statement, which must be in the first character position of an input line.
- Character strings, which can continue onto further lines: see ["Rules Governing Variables" on page 19](#).

Keywords can be abbreviated by truncation from the right, provided they are not abbreviated to the point where they become ambiguous in the particular context in which they appear.

Some keywords that appear in statement bodies are truncatable to different extents in different statements, depending on what other keywords could appear in the same positions in the particular statements. The extent to which each keyword can be safely truncated within the current release of DesignManager is shown in each statement specification by underlining of the characters that should always be retained if using the keyword.

If a keyword in a dictionary mode command statement body, in a data definition statement, or in an amendment line is truncated to an extent where it becomes ambiguous, then rather than rejecting the statement, DesignManager makes an assumption as to the intended keyword if it is reasonable to do this in the particular context. The assumption is always the safest alternative. However, users should not rely on this as no guarantees are given as to the action taken if keywords are truncated beyond the extent indicated by underlining in the statement format specifications. No such assumption is made in respect of command statement identifiers or keywords within a design mode command statement body.

If an enhancement to DesignManager results in the addition of new keywords to a member definition statement, truncation limits for previously existing keywords in that statement could change from one release to the next. Subject to the previous paragraph, ASG seeks to maintain upwards compatibility, by ensuring that if a member definition statement includes keywords that are truncated to the maximum permissible extent in a current release, it gives the same results when processed by a later release; but no guarantee can be given that this aim is always achieved. If truncation limits of such keywords are extended in the later release, then at minimum, warning messages are output when the statement is processed by the later release. In order to minimize the possibility of warning messages or unintended results occurring from later releases, users should be circumspect in truncating keywords in data definition statements and stored commands.

There are no reserved words in the DesignManager language. The language is so designed that a variable cannot be mistaken for a keyword if the user adheres correctly to the syntax defined for statements in [Chapter 3, "DesignManager Commands," on page 23](#) and [Chapter 4, "DesignManager Member Definition Statements," on page 89](#) and in the separate facilities publications. An implication of this is that words which elsewhere in context would be keywords can be used, for example, as member names.

Terminators

A terminator:

- Can be either a semicolon or a full stop (; or .).
- In a member definition statement, must be in the first character position of an input line.
- In an amendment, must start in the first character position of an input line.
- In a command statement, can either immediately follow the last preceding element of the command, or be separated from that element by a space or spaces.

When used interactively, terminators of command statements that are not associated with a following data definition statement or set of amendment lines, may be followed, with or without intervening spaces, by a further command statement contained entirely or in part on the same input line.

DesignManager statements must not be input on the same line as the terminator of a data definition statement, of an amendment, or of a command statement that must be followed by a data definition statement or set of amendment lines. The result of such input is undefined, but note that if such a terminator is not followed immediately by a space, it may not be read as a terminator.

In batch mode, each statement must start on a new line. Since command lines are always printed in batch, any characters following a recognized terminator on a command line have no effect, but being printed could be regarded as comment.

Rules Governing Variables

As stated in "[Statement Identifiers and Statement Bodies](#)" on page 16, variables can be names, integers, decimal or floating point numbers, character strings or undelimited character strings. Rules applying to variables of these different types are given here.

The rules applying to names depend on the entity to which the name applies.

A *modeling dictionary name* can be up to a maximum of 6 characters in length. It can consist of the letters of the alphabet and the numerals 0 to 9. It must commence with a letter.

A *member name* can be up to a maximum of 32 characters in length. It can consist of any of these characters:

- Letters of the alphabet
- The numerals 0 to 9
- Hyphens
- Underscores (as separate characters, not as underscores of other characters)
- £ symbols
- @ symbols
- \$ symbols (or equivalent local currency symbol, provided that it has the same internal code of hexadecimal 5B)

The first character of a member name must be a letter or a numeral or one of the symbols £, @, or \$ (or local currency symbol). If the first character is a numeral, at least one of the permitted non-numeric characters must appear in the name.

For users who have the optional Enterprise Modeling facility (Selectable Unit DSR-EM10) installed, these constraints should be observed:

- Entity names not exceeding 31 characters
- Neither entity names nor data element names starting with either of the prefix characters reserved for creating default data element names from entity names during processing of a MERGE command (see the appropriate section of *ASG-DesignManager Enterprise Modeling* and see the LHSPRE and RHSPRE parameters of the LOPT1 macro in the *ASG-Manager Products Installation in OS Environments* and *ASG-Manager Products Installation in DOS Environments*.)

Otherwise, naming conflicts may occur (see *ASG-DesignManager Enterprise Modeling*) when entities are MERGED into the workbench design area or if they are already present during processing of the MERGE command. ASG recommends that users who anticipate installation of the Enterprise Modeling facility observe the second constraint with respect to data element names.

A *member name using an extended character set* can be a string of not more than 32 printable characters enclosed within delimiters. The rules for character strings stated later in this section (with the restriction that non-printable characters must not be used) govern this form of member name. The name can consist of any printable characters, including space characters (hexadecimal 40), and can commence with any of these characters, except that the constraints given for users with the Enterprise Modeling facility should be observed.

To avoid further complicating DesignManager statement specifications, this alternative way of declaring member names, as delimited character strings, is not shown in statement formats. Therefore, subject to the limitations stated below, wherever the variable member-name appears in a statement format, it should be interpreted as offering the choice:

$$\left\{ \begin{array}{l} \text{member-name} \\ \text{'member-name'} \end{array} \right\}$$

Note: _____

Similar considerations apply where the variable is the name of a member of a specified type; for example, if *item-name* is stipulated, it can be coded as `'item-name'`.

An *alias* can be a string of not more than 79 printable characters and can commence with any of these characters. A space (hexadecimal 40) is considered as a printable character. An alias must be enclosed within delimiters, and is governed by the rules for character strings stated in this section (with the restriction that non-printable characters must not be used).

A *user name* can be up to a maximum of 32 characters in length. It can consist of any printable characters (including space characters) and can commence with any of these characters. It must be enclosed within delimiters, and is governed by the rules for character strings stated later in this section (with the restriction that non-printable characters must not be used).

An *integer* consists of the digits 0 to 9. Whether it can have a leading sign (+ or -) depends on the context, and is stated in relevant statement specifications. The number of digits in an integer must not exceed 18, and is restricted to a smaller number in some contexts.

A *decimal number* consists of an integer, optionally preceded by a sign (+ or -) and optionally followed by a decimal point and further decimal digits, but the total number of digits in the decimal number must not exceed 18. (Thus the maximum number of characters in a decimal number is 20, including a sign and a decimal point.) The last character in a decimal number must not be a decimal point.

A *floating point number* consists of:

- An optionally signed decimal number of not more than 16 digits, being the mantissa
- Optionally the letter E
- An optionally signed integer of one or two digits, being the exponent

The value of a floating point number must lie in the range:

$$5.4 \times 10^{-79} \quad \text{to} \quad .72 \times 10^{-76}$$

A *character string* is a string of any printable and/or non-printable characters, up to a maximum of 256 characters. A character string is delimited, that is, enclosed within delimiting characters which can be either single quotes (') or double quotes ("). A character string can be coded as a series of consecutive delimited character strings, which (except as stated at the end of this definition) when processed are concatenated (joined together) to form a single character string. No spaces are inserted during concatenation; allowance must be made for this fact when coding if spaces are required in the concatenated string. The number of characters in the resultant string after concatenation must not exceed 256.

Each delimited string must have the same character, single quote or double quote, as its opening delimiter and as its closing delimiter; but if delimited strings are to be concatenated, it is not necessary for them all to have the same delimiting character. The delimiting character selected for a string must not appear as a character within that string.

Each delimited string must be wholly contained on one input line, but an input line can contain two or more delimited strings. If two or more delimited strings are coded on one input line, the closing delimiter of a string must be separated by a space or spaces from the opening delimiter of the following string.

From the preceding, it can be seen that in two particular circumstances it is necessary to divide a character string into two or more delimited strings for concatenation (though use of the technique is not confined to these circumstances):

- If a character string is to contain both single quote and double quote characters.
- If a character string cannot be wholly contained on an input line.

This example illustrates the coding of delimited strings. If this passage is coded as a character string:

A string can be delimited by single quotes 'thus' or by double quotes "thus".

It could be coded in this manner:

```
'A STRING CAN BE DELIMITED BY SINGLE'  
"QUOTES 'THUS' OR" 'BY DOUBLE QUOTES "THUS".'
```

Consecutive delimited strings appearing in these common clauses (defined in ["Common Clause Specifications" on page 99](#)) are not concatenated, but are output in the same format as they were input (which imposes a limit of 252 on the number of characters they can each contain):

- ADMINISTRATIVE-DATA
- COMMENT
- DESCRIPTION
- NOTE
- QUERY

The limit of 252 characters for each string in these clauses is derived from the maximum input line length of 254 characters, less two for the delimiters of the string.

An *undelimited character string* is a string of any printable characters, up to a maximum of 254 characters. A space is not a printable character in this context, because it is identified as an element separator (see ["Statement Identifiers and Statement Bodies" on page 16](#)). The restriction is that if any of these characters: full stop (.), semicolon (;), comma (,), or right parenthesis ()) appear in the string, the string must be contained within quotes.

Amendments

An amendment comprises one or more amendment lines, and must immediately follow a dictionary mode MODIFY command. The formats of amendment lines are specified in the specifications of that command, in [Chapter 3, "DesignManager Commands," on page 23](#).

The general rules applying to the coding of command statements and data definition statements, discussed in the previous sections of this chapter, apply equally to amendments, with the exceptions that:

- The first element of each amendment line must commence in the first character position of the line.
- Each amendment line must be complete in itself, so that its coding does not continue onto a following input line.

3

DesignManager Commands

Overview

As described in ["Dictionary Mode And Design Mode" on page 13](#), DesignManager has two modes of operation: design mode and dictionary mode. In design mode, DesignManager's functions relate to the loading of data from the modeling dictionary to the workbench design area, and the manipulation and reporting of that data. In dictionary mode, DesignManager's functions relate to the input of data to the modeling dictionary, and the manipulation and reporting of that data.

The commands available in DesignManager may be grouped to reflect the two operational modes, as these three types of commands:

- Mode commands
- Dictionary mode commands
- Design mode commands

This chapter gives the specifications for all the basic DesignManager commands that are not restricted. Restricted commands are available to the Controller for controlling the usage and security of DesignManager. Restricted commands are discussed briefly in ["Controller's Commands" on page 24](#).

Note: _____

The same command identifier may be available in both modes of operation. However, the specifications and actions of such commands are dependent on the mode in which the command is issued.

The effects of DesignManager optional additional facilities on the operation and set of the basic commands are discussed in ["Effects of Optional Additional Facilities" on page 24](#). The effects of integration with the other Manager Products are discussed in ["Integrated/Non-integrated Manager Products Installations" on page 24](#).

The specifications of the mode commands are given in ["Mode Command Specifications" on page 25](#). An overview of the dictionary mode commands is given in ["Overview of the Dictionary Mode Commands" on page 25](#), and the command specifications, in alphabetical order, in ["Dictionary Mode Command Specifications" on page 28](#). An overview of the design mode commands is given in ["Overview of the Dictionary Mode Commands" on page 25](#), and the command specifications, in alphabetical order, in ["Design Mode Command Specifications" on page 73](#).

In "[Primary Command Specifications](#)" on page 28 and "[Design Mode Command Specifications](#)" on page 73, any references to ENTITY members, entity names, and the entity member type (and to the keyword ENTITIES) are applicable only for users who have the optional Enterprise Modeling facility (selectable unit DSR- EM10) installed. The keyword DATA-ELEMENTS where used refers generically to ITEM and/or GROUP member types; similarly DATA-VIEWS refers to USERVIEW and/or ENTITY member types.

General rules that apply to the coding of all DesignManager commands are given in [Chapter 2, "DesignManager Language and Coding,"](#) on page 15.

Controller's Commands

Certain DesignManager commands are restricted for use by the Controller. These commands relate to the initializing, reorganizing, and security of the modeling dictionary. They also relate to controlling the usage of DesignManager. The *ASG-Manager Products Controller's Manual* documents the Controller's commands.

Effects of Optional Additional Facilities

Some of the optional additional facilities which are available, in addition to the basic DesignManager program modules, include further commands or extensions to existing commands. Where a facility provides an additional member type (or member types), that keyword may be substituted in all DesignManager commands where the variable *optional-additional-member-type* occurs. With the exception of the integration facility, other commands and command extensions are specified in the separate publications that describe the individual facilities.

Integrated/Non-integrated Manager Products Installations

In Integrated Manager Products Installations, various facilities of the product with which DesignManager is integrated effect the facilities and operation of the basic and optional additional DesignManager facilities. For the most part, these effects are concerned with facilities related to the modeling dictionary.

This chapter documents the actions of the DesignManager commands in a non-integrated environment. The effects of integration are documented separately in [Chapter 8, "DesignManager/DataManager Integration,"](#) on page 169.

Member Names in Commands

The variable *member-name* appearing in command specifications can be alternatively coded as '*member-name*' (that is, as a delimited character string). This alternative *must* be adopted if the member-name includes characters that are not permitted in the basic member-name character set (see "[Rules Governing Variables](#)" on page 19). To avoid further complicating command statement specifications, the alternative way of declaring member names is not shown in the statement formats. Wherever the variable *member-name* appears in a command statement format, therefore, it should be interpreted as offering the choice:

$$\left\{ \begin{array}{l} \textit{member-name} \\ \text{'member-name'} \end{array} \right\}$$

Mode Command Specifications

MODE

The MODE command causes subsequent commands to process in the specified mode of operation, either design or dictionary mode. It also displays the current mode of operation.

MODE Format

$$\left. \begin{array}{l} \left\{ \begin{array}{l} \text{MODE-DESIGN} \\ \underline{\text{M1}} \end{array} \right\} \\ \left\{ \begin{array}{l} \text{MODE-DICTIONARY} \\ \underline{\text{M2}} \end{array} \right\} \\ \left\{ \begin{array}{l} \text{MODE-DISPLAY} \\ \underline{\text{MD}} \end{array} \right\} \end{array} \right\} \left\{ \begin{array}{l} ; \\ \cdot \end{array} \right\}$$

MODE Remarks

1. A MODE-DESIGN or M1 command places DesignManager in the design mode of operation. Until a subsequent MODE-DICTIONARY or M2 command is entered, DesignManager processes only design mode commands.
2. A MODE-DICTIONARY or M2 command places DesignManager in the dictionary mode of operation. Until a subsequent MODE-DESIGN or M1 command is entered, DesignManager processes only dictionary mode commands.
3. A MODE-DISPLAY or MD command causes DesignManager to display the current mode of operation, either design mode or dictionary mode.
4. At the beginning of every run, DesignManager is in the dictionary mode. Until valid dictionary mode DICTONARY and AUTHORITY commands are accepted, DesignManager will not accept any form of the MODE command.

Overview of the Dictionary Mode Commands

Separate the commands available in dictionary mode into these four groups:

- Modeling dictionary access security commands
- Modeling dictionary manipulation commands
- Modeling dictionary documentation commands
- The ENVIRONMENT command

The first three groups of commands are discussed individually in ["Modeling Dictionary Access Security Commands" on page 26](#) through ["Modeling Dictionary Manipulation Commands" on page 26](#). Full specifications of all the dictionary mode commands are given in ["Primary Command Specifications" on page 28](#) and ["Specifications of Secondary Keywords and Clauses" on page 62](#). The effects of optional additional facilities on the basic commands are discussed in ["Effects of Optional Additional Facilities" on page 24](#), and the effects of integration in ["Integrated/Non-integrated Manager Products Installations" on page 24](#). ["Member Names in Commands" on page 24](#) describes the effects of the use of the extended character set for member names.

Modeling Dictionary Access Security Commands

These commands are concerned with access to and security of a DesignManager modeling dictionary:

- DICTONARY
- AUTHORITY
- ENDDSR

DICTONARY command identifies and opens the modeling dictionary required for processing. More than one modeling dictionary may exist in any installation, and more than one can be accessed during a DesignManager run, though only one can be open at any one time during the run. In the DICTONARY command, the user declares the name of the modeling dictionary that is to open. The modeling dictionary name is established by the Controller, and is related via the job control statements to the physical files that contain the modeling dictionary's datasets. Once opened, a modeling dictionary remains open until the end of the run or until another DICTONARY command issues. An optional keyword in the command enables the dictionary to open in read-only mode.

AUTHORITY command establishes the identity of a user requiring access to the modeling dictionary. Only users authorized to do so by the Controller can access any particular modeling dictionary. In the AUTHORITY command, the user declares a password which will have been registered in the dictionary by the Controller as belonging to that user. Until an AUTHORITY command with a valid password for the dictionary currently open is received, no commands other than an ENDDSR (or ENVIRONMENT) command can be accepted by DesignManager. Thus the first two commands (except for ENVIRONMENT commands) in any DesignManager run must be a DICTONARY command followed by an AUTHORITY command.

ENDDSR command terminates a DesignManager run, and may be entered in either dictionary mode or design mode. The ENDDSR command is not necessary when DesignManager is used in batch.

Modeling Dictionary Manipulation Commands

These are the commands for manipulating a DesignManager modeling dictionary:

- ADD
- COPY
- MODIFY
- REMOVE

ADD command must always be followed immediately by a member definition statement starting on a new input line; an *empty* statement in the form of a terminator in the first character position of the next input line is acceptable. A **MODIFY** command must always be followed immediately on the next input line by amendment lines or by a terminator in the first character position of the next input line.

ADD command enables a new member's source record to be inserted into the source dataset and encoded; if no syntax errors are discovered, the encoded record is inserted into the data entries dataset. The index dataset is updated.

COPY command inserts into the source dataset a copy of the source record of an existing member of the modeling dictionary, as the source record for a new member; and updates the index dataset.

MODIFY command enables a member's source record to be amended and then encoded. If no syntax errors are discovered, the encoded record can be inserted into the data entries dataset (replacing any previous data entries record of that member). The index dataset is updated if necessary.

REMOVE command deletes a member from the modeling dictionary, provided that no other encoded members refer to that member in their data definitions.

Modeling Dictionary Documentation Commands

These are the documentation commands available in dictionary mode:

- PRINT
- LIST

PRINT command outputs the source records of selected members of the modeling dictionary, or selected parts of the source record of one member of the modeling dictionary.

LIST command lists the names of all members of the modeling dictionary, or of modeling dictionary members that come within selected categories. For each member listed, one line is output giving details of the state of the member's records. The **LIST** command can also list aliases and catalogue classifications. An alternative form of the command, **LIST HISTORY**, lists for each member or selected categories of members, when and by whom the member was inserted, last altered, and last encoded.

Dictionary Mode Command Specifications

Primary Command Specifications

This section outlines the primary command specifications for the four types of dictionary mode commands.

ADD

The ADD command is a modeling dictionary manipulation command. Use the ADD command to insert a member's source record into the modeling dictionary, to generate an encoded record from the newly inserted record, and to insert the encoded record into the data entries dataset.

ADD Format

```
ADD member-name [NUMBER integer] { ; }  
                                     { . }
```

where:

member-name is the name of a modeling dictionary member for which a source record is to be inserted.

integer is an unsigned integer in the range 1 to 10,000, being the line number for the first line and the increment for subsequent line numbers of the member's source record.

ADD Remarks

1. A member definition statement must immediately follow the ADD command. The lines of the member definition statement must be unnumbered. For the formats of member definition statements, see [Chapter 2, "DesignManager Language and Coding," on page 15](#) and [Chapter 4, "DesignManager Member Definition Statements," on page 89](#).
2. Member names must conform to the rules stated in ["Rules Governing Variables" on page 19](#). Users who have the optional additional Enterprise Modeling facility (Selectable Unit DSR-EM10) installed should observe the indicated constraints with respect to entity and data element names.
3. If no source record for *member-name* exists in the modeling dictionary, a source record containing the member definition statement is created for *member-name*. In the source record, the lines of the member definition statement are numbered as stated in [remark 5](#) or [remark 7](#).
4. If a source record for *member-name* exists in the modeling dictionary, this error message outputs, the command and the associated member definition statement reject, and processing continues with the next command:

```
member-name SOURCE ALREADY PRESENT
```

5. If NUMBER is stated in the command and is followed by a valid integer, the lines in the member's source record are numbered in increments specified by integer, commencing with the number specified by integer. If the integer is signed but is not more than 10,000, the sign is removed, this warning message outputs, and the integer is otherwise treated as valid:

INVALID INTEGER - SIGN REMOVED

6. If the increment specified by integer is such that line numbers extend into more than five digits, only the last five digits of each line number are output in any printout or display of the source record; except in response to a DISPLAY amendment line (see ["MODIFY" on page 44](#)), when all eight digits display.
7. If NUMBER is not stated in the command, or if NUMBER is stated in the command but is not followed by a valid integer, the lines in the member's source record are numbered in increments of 100, commencing with 100.
8. An encoded record generates if the source record is successfully inserted.
9. If the source record is syntactically correct and no dummy data entries record exists for member-name, the encoded record is inserted into the data entries dataset.
10. If the source record is syntactically correct and a (dummy) data entries record exists for member-name, the encoded record replaces the existing record in the data entries dataset. The "used by" pointers in the existing data entries record are copied to the newly encoded record before the existing record is deleted.
11. If, during the generation of the encoded record, the source record is found to have errors of syntax, diagnostic messages are output in respect of the source record, and the encoded record rejects; any (dummy) data entries record that may exist for member-name is retained.
12. In addition to syntax checks, these checks are performed during the encoding phase of the command:
 - That no reference from the member being encoded is to an encoded or dummy member that is of a wrong member type in the context of the reference. (For example, if the member being encoded is a GROUP, its CONTAINS clause must not refer to an encoded or dummy member that is a USERVIEW.) If any member to which the member being encoded refers has an encoded or dummy data entries record of a member type that is invalid in the context of the reference, this message outputs, and the encoding rejects. Validation of the remainder of the data definition continues:

member-type member-name IS NOT A VALID MEMBER-TYPE IN THIS CONTEXT
 - That there is no change in the member type of the member being encoded that would be invalid in the context of any existing reference to that member. (For example, if an encoded GROUPS's CONTAINS clause referred to the member, the member could not be encoded as a USERVIEW.) If there is a change in the member's member type that would make existing references to the member from other encoded members invalid, this message outputs, and the encoding rejects:

CHANGE OF MEMBER-TYPE NOT ALLOWED - CHECK USAGE

This check is the last one performed. It is omitted if any errors are discovered earlier in the encoding phase of the command.

13. If the encoded record is successfully inserted, it includes pointers to the data entries records of all members to which member-name refers directly. If member-name refers to a member for which no data entries record yet exists, a dummy data entries record is created for that member. *Used by* pointers are inserted into the data entries records (including dummies) of all members to which member-name refers, to point back to member-name.
14. If member-name already had a dummy data entries record and the encoding of member-name results in a permissible change of its member type (see [remark 12](#)), this change is reflected in the *uses* tables of all members that refer to member-name.
15. The index dataset is updated.
16. The automatic error recovery system treats ADD as an updating command. If a failure occurs during the updating of the source dataset, the automatic error recovery system restores the modeling dictionary to the state it was in immediately before the ADD command was actioned. If a failure occurs during the encoding phase of the command, that is, after this message outputs, the automatic error recovery system restores the modeling dictionary to the state it was in at the time that the message was output:

```
member-name SUCCESSFULLY INSERTED
```

Note: _____

In the latter case, the user should issue a MODIFY command in a subsequent run.

ADD Examples

These examples illustrate two ADD commands with their associated data definition statements. In the first example, lines in the resulting source record are numbered in increments of 100; in the second example, lines in the resulting source record are numbered in increments of 200:

```
ADD DATE ;
GROUP
CONTAINS DAY, MONTH, YEAR
;
```

```
ADD OFFICE-NUMBER NUMBER 200 ;
ITEM
ALIAS OFFICE-LOCATION
CATALOGUE LOCATION
SEE BUILDING-ID
;
```

AUTHORITY

The AUTHORITY command is a modeling dictionary access security command. Use this command to identify the user next requiring access to the modeling dictionary; or to identify the Master Operator.

AUTHORITY Format

```
AUTHORITY 'password' { ; }  
                  { . }
```

where *password* is a character string of up to 8 printable or non-printable characters, being the identification issued by the Controller to the particular user or to the Master Operator.

AUTHORITY Remarks

1. A DICTONARY command must open a modeling dictionary in order for an AUTHORITY command to be accepted.
2. An AUTHORITY command, quoting a password which is registered in the modeling dictionary by the Controller, must be input by the user or the Master Operator before any other command, except a DICTONARY or ENDDSR command, is accepted.
3. If the user's authority to access the modeling dictionary is established by acceptance of the AUTHORITY command, that authority remains in force until a further AUTHORITY command is input or the run terminates.
4. If the Master Operator's authority to process the modeling dictionary is established by acceptance of the AUTHORITY command, only those commands pertaining to the Master Operator are accepted, until a further AUTHORITY command is input or the run terminates.
5. If an AUTHORITY command is input, the effect of any AUTHORITY command accepted earlier in the run is terminated irrespective of whether the new AUTHORITY command is accepted or rejected.
6. It follows that if an AUTHORITY command with an invalid password is input, no command other than a DICTONARY or ENDDSR command is accepted until an AUTHORITY command with a valid password is input.
7. As the password issued to the user or to the Master Operator is part of the system of security of access to the modeling dictionary, the user or the Master Operator should comply strictly with any procedures established by the Controller to ensure the privacy of passwords.
8. When DesignManager is used in batch, if the password is input on a different line from the AUTHORITY command identifier, the password is not printed. This makes the use of passwords in batch environments more secure.
9. When DesignManager is used interactively from a hard copy terminal, printing of the password can be suppressed if the terminal is switched to full duplex operation when the password is input.

10. Passwords are recorded in the modeling dictionary as 8-character fields, left-justified with trailing spaces (hexadecimal 40). For example, the passwords 'ABC' and 'ABC ' are treated by DesignManager as identical if the last three characters of 'ABC' are space characters (but not if they are other non-printing characters). The passwords 'ABC' and 'AB C', however, are not treated as identical.

AUTHORITY Examples

In a batch environment, the password in the second example would not print, increasing security.

```
AUTHORITY ' PMGR' ;
```

```
AUTHORITY  
'SYS27' .
```

COPY

The COPY command is a modeling dictionary manipulation command. Use the COPY command to insert as the source record of a new member of the modeling dictionary, a copy of the source record of an existing member.

COPY Format

```
COPY member-name TO member-name-2 [REPLACE] { ; }  
{ . }
```

where:

member-name is the name of the existing member whose source record is to be copied.

member-name-2 is the name by which the new member, whose source record is to be a copy of member-name's source record, is to be known.

COPY Remarks

1. Member names must conform to the rules stated in ["Rules Governing Variables" on page 19](#). Users who have the optional additional Enterprise Modeling facility (Selectable Unit DSR-EM10) installed should observe the indicated constraints with respect to entity and data element names.
2. The COPY command inserts a copy of the source record of member-name into the source dataset as a record for a new member whose name is determined by member-name-2. The index dataset is updated.
3. If a source record already exists for member-name-2 and REPLACE is not present in the command, this error message outputs, the COPY command rejects, and processing continues with the next command:

```
member-name-2 SOURCE ALREADY EXISTS
```

4. If a source record already exists for member-name-2 and REPLACE is present in the command, a copy of member-name's source record replaces member-name-2's existing source record.
5. The automatic error recovery system treats COPY as an updating command.

COPY Example

The command COPY ARTICLE-NUMBER TO COMPONENT-ARTICLE-NUMBER REPLACE; copies the source record of ARTICLE-NUMBER into a new source record for COMPONENT-ARTICLE-NUMBER. If a source record for COMPONENT-ARTICLE-NUMBER already exists, it is replaced by the copied record.

DICTIONARY

The DICTIONARY command is a modeling dictionary access and security command. Use the DICTIONARY command to identify the modeling dictionary to open for processing.

DICTIONARY Format

```
DICTIONARY dictionary-name [ { READ } ] [SYSNAME file-name]
                        { UPDATE }
[IBUF n] [SBUF n] [DBUF n] { ; }
                        { . }
```

where:

dictionary-name is the name of a modeling dictionary established by the Controller.

file-name is the logical file name (ddname) used in OS job control statements to indicate the external dataset name (physical file name) of the modeling dictionary's index dataset.

n is an unsigned integer in the range 1 to 255.

DICTIONARY Remarks

1. Only one modeling dictionary can be open at any one time in a DesignManager run. When a valid DICTIONARY command is input, any modeling dictionary currently open is closed, and the modeling dictionary specified by dictionary-name opens for processing.
2. If an invalid DICTIONARY command is input, any modeling dictionary currently open is closed. It is then necessary to input a valid DICTIONARY command before further processing can take place.
3. Except for an ENDDSR command, DICTIONARY is the only command that can be accepted without the prior acceptance of an AUTHORITY command. This is because the security information on which acceptance of an AUTHORITY command is based is contained within the modeling dictionary itself, and hence is not accessible until the modeling dictionary opens.

4. Whenever a DICTONARY command is accepted, it must be followed by an AUTHORITY command valid for the modeling dictionary opened by that DICTONARY command, before any commands to process that modeling dictionary are accepted.
5. If neither READ nor UPDATE is stated in the DICTONARY command, then:
 - In all non-CMS environments, the modeling dictionary opens in update mode, unless the value of the UPDATE parameter of the DCUST installation macro is tailored to NO, in which case the modeling dictionary opens in read-only mode. (The specifications of DCUST are given in *ASG-Manager Products Installation in OS Environments* and in *ASG-Manager Products Installation in DOS Environments*.)
 - In CMS environments, the modeling dictionary opens in read-only mode (irrespective of the value of the UPDATE parameter of DCUST).
6. If the READ keyword is present in the DICTONARY command, then the modeling dictionary opens in read-only mode (irrespective of the value of the UPDATE parameter of DCUST) in any environment.
7. If a modeling dictionary is opened in read-only mode, the error recovery dataset and if logging is applied to the modeling dictionary, the log dataset, are not opened. Processing of all commands is marginally faster when the modeling dictionary is open in read-only mode than when it is open in update mode.
8. If UPDATE=NO was declared in the installation macro DCUST, a DICTONARY command with an UPDATE keyword is not accepted; a valid DICTONARY command must be input before further DesignManager processing can take place.
9. If the value of the UPDATE parameter of DCUST is YES, an UPDATE keyword in a DICTONARY command is accepted but, except in CMS environments, is superfluous. In CMS environments, the DICTONARY command must include the UPDATE keyword if any updating command is to be used.
10. The SYSNAME clause is available for OS environments to enable a standard job control procedure to be catalogued to execute DesignManager operating on any modeling dictionary, where an installation has more than one DesignManager modeling dictionary. DesignManager recognizes dictionary-name, and outputs that name in page headings and in any messages that refer to the modeling dictionary by name. DesignManager relates dictionary-name to the logical name in the OS job control DD statements. A substitution would need to be defined in the EXEC job control statement for the physical file name (DSN name) stated in the DD statements

For example, if the job control included the statements:

```
//DDICT DD DSN=DICT . INDEX, etc.  
//DDICTD DD DSN=DICT . DATA . ENTRIES, etc.  
//DDICTS DD DSN=DICT . SOURCE, etc.  
//DDICTE DD DSN=DICT . RECOVER, etc.  
//DDICTJ DD DSN=DICT . LOC, etc.
```

and it was required to access a modeling dictionary named DPOOL, for which the DSN names were POOL.INDEX, POOL.DATA.ENTRIES, POOL.SOURCE, POOL.RECOVER and POOL.LOG respectively, then it would be necessary to include the substitution DICT=POOL in the EXEC statement. The relevant DICTIONARY command would be as in the second example in ["DICTIONARY Examples" on page 36](#).

11. The IBUF, SBUF, and DBUF clauses override respectively the values of the IBUF, SBUF, and DBUF parameters of the DCUST installation macro. They specify the number of additional input/output buffers that are to be used respectively for the index, source, and data entries datasets. (See *ASG-Manager Products Installation in OS Environments* or *ASG-Manager Products Installation in DOS Environments*.)
12. The sequence of actions when a DICTIONARY command is received, and the checks performed to determine whether the command is valid, are these:
 - A check is made that there are job control statements for the modeling dictionary's index, source, and data entries datasets, for the error recovery dataset (unless READ is stated in the command and/or the value of the UPDATE parameter of DCUST is NO) and for the log dataset (if logging is applied to the modeling dictionary and the modeling dictionary is being opened in update mode); and that the datasets are on line. If the check fails, this message outputs:

```
type DATA SET NOT ALLOCATED/CREATED
```

where *type* is INDEX, DATA, SOURCE, RECOVERY, or LOG.
 - A check is made that the control block record in each dataset of the modeling dictionary corresponds with the dictionary-name specified in the command. If the check fails, this message outputs:

```
name NOT FOUND IN DICTIONARY DATASET CONTROL BLOCK
```

where *name* is the name of the required dataset with which a control block fails to correspond. This check is made to ensure that datasets from different modeling dictionaries are not brought together.
 - A check is made that the date and time of creation of each dataset of the dictionary agrees with the index creation date and time. If the check fails, this message outputs:

```
name NOT CREATED AT SAME DATE/TIME AS INDEX
```

where *name* is the name of the dataset whose date and/or time of creation differs from those recorded in the index. This check is made to ensure that different versions of datasets of the same modeling dictionary are not incorrectly associated (for example, that old and new datasets are not wrongly brought together after a restore) and that an unrelated dataset having the same name as a modeling dictionary dataset is not accessed in place of a modeling dictionary dataset.
 - The automatic error recovery system recovers the modeling dictionary if necessary.
 - A check is made that the update control numbers of the source, data entries, and error recovery datasets of the modeling dictionary agree with the index control number, and that the control number of the log dataset agrees with that recorded for it in the index dataset. If the check fails, this message outputs:

```
name DOES NOT HAVE SAME UPDATE CONTROL NUMBER AS INDEX
```

where *name* is the name of the dataset concerned, or, for the log dataset output:

```
TRANSACTION CONTROL NUMBER MISMATCH
```

This is a further check that different versions of the datasets are not wrongly associated after a restore (for example, that a restore of a single dataset has not taken place).

- If all checks succeed, the DICTONARY command is accepted as valid.

13. If, when the modeling dictionary opens, the alternate area of the log dataset requires archiving, this warning message outputs:

```
NO ALTERNATE LOG AREA AVAILABLE
```

Note: _____

If this message appears, inform the Controller immediately. Failure to take archiving action could result in the present log area becoming full, in which case use of the modeling dictionary is inhibited.

DICTIONARY Examples

This example would open a modeling dictionary named DDICT:

```
DICTIONARY DDICT ;
```

This example would open a modeling dictionary named DPOOL, when the logical name by which it was identified in the OS job control DD statements was DDICT (See further in [remark 10 on page 34](#)):

```
DICTIONARY DPOOL SYSNAME DDICT;
```

This example would open a modeling dictionary named DDICT in read-only mode, with 10 additional input/output buffers each for the source and data entries datasets (where the DCUST macro specified different numbers of additional input/ output buffers for those datasets):

```
DICTIONARY DDICT READ SBUF 10 DBUF 10.
```

Figure 2 contains a summary of the modes in which modeling dictionaries are opened in varying combinations of DCUST UPDATE parameter settings and DICTIONARY command keywords.

Figure 2. Modeling Dictionary Opening Modes

| Modeling Dictionary Opening Modes: Interaction of the DCUST Macro, DICTIONARY Command Keywords and Environments | | | | |
|--|------------|--------|------------|------------|
| Environment: | Non CMS | CMS | Non CMS | CMS |
| DCUST Macro DICTIONARY Command; | UPDATE=YES | | UPDATE=NO | |
| DICTIONARY; | Update | Read | Read | Read |
| DICTIONARY READ; | Read | Read | Read | Read |
| DICTIONARY UPDATE; | Update | Update | Not opened | Not opened |

ENDDSR

The ENDDSR command is a modeling dictionary access and security command. Use the ENDDSR command to terminate a DesignManager run.

ENDDSR Format

```
ENDDSR { ; }
        { . }
```

ENDDSR Remarks

1. A DesignManager run terminates on acceptance of this command. Under TSO it offers an alternative to /*.
2. The command is not necessary (but is accepted) when DesignManager is used in batch.
3. ENDDSR may be input when DesignManager is in either dictionary mode or design mode.

ENDDSR Examples

```
ENDDSR .
```

```
END ;
```

ENVIRONMENT

The ENVIRONMENT command is a modeling dictionary access and security command. Use the ENVIRONMENT command to output the release number of each installed Manager Product, together with a list of all the fixes that have been applied to each product in this installation.

ENVIRONMENT Format

```
ENVIRONMENT { ; }  
            { . }
```

ENVIRONMENT Remarks

1. The ENVIRONMENT command is accepted whether or not a dictionary is currently open.
2. When the ENVIRONMENT command is accepted, the number of the DesignManager release currently in use is output, followed by a list of all the fixes that have been applied to that release in this installation.
3. If the command is used in an integrated Manager Product installation, then the above information is output for each of the Manager Products in the installation.
4. The ENVIRONMENT command is intended to be used as a diagnostic aid. The output from the command can be checked against current software notices to determine whether all appropriate fixes have been applied.
5. If it becomes necessary to seek assistance with a problem in a Manager Product, your local Product Support Office will ask for the output from an ENVIRONMENT command. The output from the command should therefore be obtained before requesting support.

LIST

The LIST command is a modeling dictionary documentation command. Use the LIST command to list all or list selected categories of:

- Members of the modeling dictionary, with information concerning their records
- Aliases
- Catalogue classifications

LIST Format

```
LIST [HISTORY] [UNVERIFIED] [selection]  
     [ALPHABETICALLY] [name-related-selection]  
     [time-and-user-related-selection]  
     [DETAILS-ONLY] { ; }  
                   { . }
```

where *selection* is of this format:

```
[MEMBERS member-name [, member-name] ...]  
[category]
```

where:

member-name is the name of a member of the modeling dictionary.

category is:

| | | | | |
|---|--------------------|--|---|----------------------------------|
| { | [<u>DUMMY</u>] | <i>index-name-type</i> | } | |
| | <u>EXCEPT</u> | [, <i>index-name-type</i>] ... | | |
| { | <u>DUMMYS</u> | [<u>EXCEPT</u> <i>index-name-type</i> | } | |
| | <u>DUMMIES</u> | | | [, <i>index-name-type</i>] ...] |
| | <u>INDEX-NAMES</u> | | | |
| } | <u>SOURCES</u> | | | |

where *index-name* is:

| | | |
|---|---|---|
| { | <u>ITEMS</u> | } |
| | <u>GROUPS</u> | |
| | <u>DATA-ELEMENTS</u> | |
| | <u>DATA-VIEWS</u> | |
| | <u>USERVIEWS</u> | |
| | <u>ENTITIES</u> | |
| | <u>VIEWSETS</u> | |
| | <i>optional-additional-member-types</i> | |
| | <u>ALIASES</u> | |
| | <u>CATALOGUES</u> | |
| | <u>CATALOGS</u> | |

where:

optional-additional-member-types refer to additional member types provided in optional additional DesignManager facilities (see [Chapter 8, "DesignManager/DataManager Integration," on page 169](#)).

name-related-selection is the FROM clause and/or the TO clause, or the ONLY clause, and/or the WHEN clause, as specified in "[Specifications of Secondary Keywords and Clauses](#)" on page 62.

time-and-user-related-selection is as specified in "[Specifications of Secondary Keywords and Clauses](#)" on page 62.

LIST Remarks

1. In these remarks, use the term *index-names* to cover this type of data:
 - Member names
 - Aliases and catalogue classifications of encoded members
2. If UNVERIFIED is present in a LIST command, index-names process (subject to selection, name-related-selection, and time-and-user-related-selection) only if they satisfy one of these conditions:
 - They have a source record but no encoded recorded
 - That have a source record that has been altered since it was last encodedUNVERIFIED is meaningless in respect of the index-names-types ALIASES, CATALOGS, and CATALOGUES.
3. If selection is present in a LIST command, the action of the command is restricted to those index-names that satisfy a condition or conditions specified by selection. If UNVERIFIED and selection are both present in the command, index-names determined by selection are processed if they are also unverified (see [remark 2](#) above).
4. If the selection MEMBERS member-name <, member-name> . . . is stated in the command, members whose names are listed in the MEMBERS clause are processed. Any number of member names can be listed in the clause; each name, except the first in the list, must be preceded by a comma and can optionally be preceded by spaces.
5. If a category selection is stated in the command, the action of the command is restricted to those index-names that fall within that category. If category is stated in addition to a MEMBERS clause, only those index-names that appear in the list of members and that are also of the specified category are processed.
6. If two or more index-name-type categories are listed in the command, each except the first in the list must be preceded by a comma. Entries in the list may additionally be separated by spaces. Index-names process if they are of any of the types listed.
7. If index-name-type or an index-name-type list is stated and is not preceded by the keyword EXCEPT, index-names are processed if they are of any of the index-name-types stated. If the keyword EXCEPT precedes the index-name-type or index-name-type list, all index-names that are not of any of the specified types are processed.
8. For index-name-type categories other than ALIASES AND CATALOGUES or CATALOGS, only members with encoded or dummy data entries records are processed. (The member type of a member that has only source and index records cannot be determined by DesignManager.)
9. The category ALIASES covers all aliases of all encoded members.
10. The category CATALOGUES or CATALOGS covers all classifications under which any encoded members are catalogued.

11. If an index-name-type category or a list of index-name-type categories is preceded by the keyword DUMMY (which must not be truncated), then members having dummy data entries records of the specified type(s) are processed. Such members can have source records that have not been encoded. DUMMY is accepted in conjunction with ALIASES and/or CATALOGUES or CATALOGS but has no effect on the processing of those categories.
12. The category DUMMYS or DUMMIES includes only members for which dummy data entries records have been set up as a result of the encoding of other members which refer to these members; such members can have source records that have not been encoded. The truncation limits of these keywords as defined in ["LIST Format" on page 38](#), should be noted.
13. If the category DUMMYS or DUMMIES is followed by an EXCEPT clause, then members that have dummy data entries records are processed provided that the dummy records are not of any member type that is specified in the EXCEPT clause.
14. The category INDEX-NAMES covers all index-names as defined in [remark 1 on page 40](#). If INDEX-NAMES is followed by an EXCEPT clause, then index-names are excluded from processing if they are of any of the types stated in the EXCEPT clause.
15. The category SOURCES includes only members that have a source record but no encoded record; such members can have dummy data entries records.
16. The category DATA-ELEMENTS covers all members of the types ITEM and GROUP.
17. The category DATA-VIEWS covers all members of the types USERVIEW and ENTITY.
18. The keyword ALPHABETICALLY is meaningful only in the context of a MEMBERS clause selection. If ALPHABETICALLY is present in the command, those selected index-names are processed in alphanumeric order. If ALPHABETICALLY is not present in the command, those selected index-names are processed in the order in which they appear in the list. If a MEMBERS clause is not present in the command, the selected categories are processed in alphanumeric order of index-name; in these circumstances, ALPHABETICALLY, if present, is accepted but has no effect.
19. Further restrict the action of a LIST command by name-related-selection and/or time-and-user-related-selection. Name-related-selection and time-and-user-related-selection are defined in ["Specifications of Secondary Keywords and Clauses" on page 62](#).
20. If the keyword DETAILS-ONLY is present in the command, only the detail lines of the list are output. The command output heading(s) and total lines are suppressed. Paper throws before and after the command are also suppressed, but not paper throws within the list.
21. If no selection is stated in the command, a selection of all members is assumed; that is, subject to the UNVERIFIED keyword, name-related-selection and time-and-user-related-selection, a list of all members, but not of aliases or catalogue classifications.

22. If the LIST command does not include the keyword HISTORY, then a list is output containing one line for each index- name classification processed, with these column headings:

MEMBER NAME. This column of the list contains, in alphanumeric order (except when a MEMBERS clause selection is present in the command and ALPHABETICALLY is not stated), the names of the members, aliases, or catalogue classifications processed.

TYPE. This column contains the member type, or one of the words ALIAS or CATALOGUE, as appropriate. If an alias or catalogue classification is more than 32 characters, the word ALIAS or CATALOGUE respectively is moved to the next line.

USAGE. This column shows:

- For a member having an encoded or dummy data entries record, the number of direct references to this member from other members. For a member having no data entries record, the column is blank.
- For an alias, the number of encoded members to which the alias relates.
- For a catalogue classification, the number of encoded members catalogued under that classification.

CONDITION. For an alias or a catalogue classification, this column is blank. For a member, it indicates what records (additional to the index record) exist for the member, thus:

- If a source record exists for the member, SCE is stated.
- If an encoded data entries record exists for the member, ENC is stated, SCE and ENC are both stated if appropriate.
- If a dummy data entries record exists for the member, DUM is stated. SCE and DUM are both stated if appropriate.
- If the member has:
 - A dummy data entries record
 - A source record but no encoded data entries record
 - A source record that has been changed in any way since the member was last encoded then the entry in this column is flagged by a preceding asterik (*)

23. At the end of the list, a count is given of:
- The number of each type of member in the list
 - The number of aliases listed
 - The number of catalogue classifications listed

24. If the LIST command includes the keyword HISTORY, a list is output containing in respect of each index-name processed:
 - A line containing the member name, alias, or catalogue classification.
 - A group of lines stating:
 - MEMBER-TYPE with the member type, or ALIAS or CATALOGUE
 - For members, if appropriate, the date and time and by whom the member was inserted into the modeling dictionary.
 - For members, if appropriate, the date and time, and by whom the member was last altered.
 - For members, if appropriate, the date and time, and by whom the member was last encoded.
 - For members with dummy data entries records, four asterisks (****) and the word DUMMY, followed by the date and time, and by whom the dummy record was created as a result of encoding a referring member.
 - For aliases and catalogue classifications, the date and time, and by whom the alias or catalogue classification record was created by the encoding of a relevant member.
25. At the end of the history list, a count is given of the number of entries in the list. The count includes all members, aliases, and catalogue classifications listed, but is expressed as a total number of members.
26. Elements present in a LIST command must be in the order shown under ["LIST Format" on page 38](#).

LIST Examples

This example results in the output of a list of all members with dummy data entries records that were created as dummy item records:

```
LIST DUMMY ITEMS .
```

This example results in the output of a list of all USERVIEW members that have only a source record in the modeling dictionary or have a source record that has been altered since the encoded record was created:

```
LIST UNVERIFIED USERVIEWS ;
```

This example results in the output of a history list of all members whose names began with a character other than a letter or a numeral, together with those whose names began with the letter A:

```
LIST HISTORY TO 'A' ;
```

This example results in the output of a consolidated list of all members, aliases, and catalogue classifications. No headings or totals would be printed and paper throws before and after the command would be suppressed:

```
LIST INDEX-NAMES DETAILS-ONLY .
```

This example results in the output of a consolidated list of all members, aliases, and catalogue classifications whose names included the word EMPLOYEE:

```
LIST INDEX-NAMES WHEN ANY = 'EMPLOYEE' ;
```

MODIFY

The MODIFY command is a modeling dictionary manipulation command. Use the MODIFY command to amend the source record of a member of the modeling dictionary, and to encode the amended record.

MODIFY Format

Format of the MODIFY command and associated amendment lines.

```

MODIFY member-name [RENUMBER [integer]] { ; }
                                           { . }

[ { m } [source-line]
  { LINE }
  DELETE { m [THRU n] }
          { TO }
          { LINE }
  MOVE { m [THRU n] } { BEFORE } { r }
  COPY { LINE }      { TO }      { LINE }
          { AFTER }
  CHANGE [ { m [THRU n] } [ALL] 'string' [TO] 'string-2' [PRINT]
          { LINE }
  FILL { { m [THRU n] } [COLUMNS a [THRU b]] 'string' [PRINT]
        { LINE }
        { COLUMNS a [THRU b] }
  BLANK { { m [THRU n] } [COLUMNS a [THRU b]] [PRINT]
         { LINE }
         { COLUMNS a [THRU b] }
  DISPLAY [ { m [THRU n ] ]
           { END }
           { LINE }
  LOCATE { [m [THRU n ] ] 'string'
          { END
          { NEXT ['string']
          { ALL 'string' }
  RENUMBER m THRU n [FROM v] IN i
  ACCEPT
  QUIT
  END
  ;
  . } ] ... ]

```

where:

member-name is the name of a modeling dictionary member whose source record and data entries record are to be amended.

integer is an unsigned integer in the range 1 to 10,000, being the new line number for the first line and the increment for subsequent line numbers of the member's altered source record.

$\left. \begin{array}{l} . \\ m \\ n \\ r \\ v \end{array} \right\}$ are unsigned integers of not more than eight digits in the range 1 to 16,777,215, being source record line numbers. If present, *n* must be greater than *m*.

source-line is a member definition statement line appropriate to the member type; that is, a line conforming to the specifications stated in [Chapter 2, "DesignManager Language and Coding," on page 15](#), and to the specifications stated in [Chapter 4, "DesignManager Member Definition Statements," on page 89](#) for the member type concerned.

$\left. \begin{array}{l} . \\ string \\ string-2 \end{array} \right\}$ are character strings of 1 to 254 printable characters, subject to limits on their length imposed by the line length of the input device and the other elements present in the input line (See [remark 16 on page 49](#)).

$\left. \begin{array}{l} a \\ b \end{array} \right\}$ are unsigned integers in the range 1 to 254, being character positions in a source record line. If present, *b* must be greater than *a*.

i is an unsigned integer in the range 1 to 10,000.

MODIFY Remarks

1. The MODIFY command is optionally followed by an amendment line or a series of amendment lines. Each amendment line must be input as a separate line. The amendment line or the last line in the series of amendment lines must be followed by a line containing a terminator in the first character position, or by END, ACCEPT, or QUIT commencing in the first character position. If there are no amendment lines associated with the command, the command must be followed by a line containing a terminator in the first character position, or by END or QUIT commencing in the first character position. (END used as defined in this remark is treated by DesignManager as an amendment terminator; it does not terminate a DesignManager run.) No DesignManager command can be input on the same line as an amendment terminator.
2. Subject to qualifications stated in [remark 13 on page 48](#), [remark 14 on page 49](#), [remark 59 on page 58](#) and [remark 60 on page 59](#), if a source record for member-name exists, a new version of the record, incorporating the changes specified by the MODIFY command and its associated amendment lines, is written in the source dataset. An encoded record for the member is then generated from the new version of the source record; the encoded record includes pointers to the data entries records of all members to which member-name refers directly. The space occupied by the previous version of the source record is released when the new source record is successfully encoded.

3. If the source record is syntactically correct and the member has no existing data entries record, the newly encoded record is inserted into the data entries dataset. If member-name refers to a member for which no data entries record yet exists, a dummy data entries record is created for that member. *Used by* pointers are inserted into the data entries records (including dummies) of all members to which member-name refers, to point back to member-name. The index dataset is updated.
4. If the source record is syntactically correct and a data entries record already exists for member-name, the newly encoded record replaces the existing record. If the existing record contains any *used by* pointers indicating other members of the modeling dictionary, these pointers are copied to the newly encoded version of the record before the existing record is deleted. *Used by* pointers are inserted in or deleted from the data entries records of other members, and/or dummy records are created, where necessary to reflect any differences between the references contained in the old and new versions of member-name's data entries record. If these adjustments result in any dummy data entries record being left with no *used by* pointers, such dummy records are deleted, as they no longer serve any purpose.
5. If member-name already had an encoded or dummy data entries record and the encoding of member-name results in a permissible change of its member-type (see [remark 8](#)), this change is reflected in the *uses* tables of all members that refer to member-name.
6. The index dataset is updated in respect of any new member names resulting from the creation of dummy members.
7. If, during the generation of the encoded record, the amended source record is found to have errors of syntax, diagnostic messages are output in respect of the source record, and the newly encoded record is rejected. Any previously existing data entries record for member-name is retained. The source record is retained in its amended form. The previous version of the source record is also retained if it has an encoded data entries record, or is released if it has no encoded data entries record.
8. In addition to syntax checks, these checks are performed during the encoding stage of the MODIFY command:
 - That no reference from the member being encoded is to an encoded or dummy member that is of a wrong member type in the context of the reference. (For example, if the member being encoded is a GROUP, its CONTAINS clause must not refer to an encoded or dummy member that is a USERVIEW.) If any member to which the member being encoded refers, has an encoded or dummy data entries record of a member type that is invalid in the context of the reference, this message is output and the encoding rejects:

```
member-type member-name IS NOT A VALID MEMBER-TYPE IN  
THIS CONTEXT
```

Validation of the remainder of the modified data definition continues.
 - That there is no change in the member type of the member being encoded that would be invalid in the context of any existing reference to that member. (For example, if an encoded GROUP's CONTAINS clause referred to the member, the member could not be encoded as a USERVIEW.) If there is a change in the member's member type that would make existing references to the member from other encoded members invalid, this message is output and the encoding is rejected:

CHANGE OF MEMBER-TYPE NOT ALLOWED - CHECK USAGE

This check is the last one performed. It is omitted if any errors are discovered earlier in the processing of the command.

9. If no source record exists for member-name, this error message is output, the command and the associated amendment lines are rejected, and processing continues with the next command:

member-name HAS NO SOURCE TO BE ALTERED

10. If RENUMBER is stated in the command and is not followed by a valid integer, the lines in the member's new source record are renumbered in increments of 100, commencing with 100. If RENUMBER is stated in the command and is followed by a valid integer, the lines in the member's new source record are renumbered in increments specified by integer, commencing with the number specified by integer. If the integer is signed but is not more than 10,000, the sign is removed, this warning message outputs, and the integer is otherwise treated as valid:

INVALID INTEGER - SIGN REMOVED

11. If lines in the member's new source record are renumbered, and the increment is such that line numbers extend into more than five digits, only the last five digits of each line number are output in any print-out or display of the new record, except in response to a DISPLAY amendment line, when all eight digits display.
12. If RENUMBER is not stated in the command, lines in the member's new source record are numbered as in the old source record or as left by the processing of the input amendment lines (see [remark 13](#) on this page through [remark 61 on page 59](#)).
13. The syntax checking of the command performs before its associated amendment lines are read. If there are any syntax errors in the command, all following amendment lines up to the amendment terminator are rejected. If there are no syntax errors in the command, then when the command terminator is read, member-name's source record copies into an area of DesignManager working storage. Member-name then releases to be available for access or update by other users, while the amendment lines are processed to operate upon the working storage copy source record. If the processing of an amendment line results in an insertion of lines in the copy source record that necessitates the renumbering of the lines following the insertion (see [remark 33 on page 52](#) through [remark 35 on page 53](#) below), such renumbering is performed before the next amendment line is processed; subsequent amendment lines operate upon the copy source record as amended and renumbered. The user must allow for any such renumbering when specifying later amendment lines to be processed by the same command; the DISPLAY amendment line (see [remark 49 on page 56](#) through [remark 51 on page 57](#)) can be used, in conjunction if necessary with QUIT (see [remark 60 on page 59](#)), to ascertain whether amendments are correctly specified. When the amendment terminator is read:

- DesignManager checks whether another amended version of the member has been written to the source dataset by another user while this command was processed. If another amended version is written while this command processes, then the version resulting from this command rejects and an error message is output:

member-name SOURCE MEMBER UPDATED BY ANOTHER USER

- If no other amended version of member-name is written to the source dataset while this command processes, the amended record is written to the source dataset. While the record is written to the source dataset, access to member-name by other users is prevented.
 - If a RENUMBER clause is present in the command, the lines of the amended source record are renumbered as stated in [remark 10 on page 48](#) while written to the source dataset.
14. If any error condition arises during the processing of amendment lines, then:
- In an interactive processing run, the user can correct the error by submitting a further amendment line and processing of the command can continue; or the user can abandon the command by submitting a QUIT amendment line (see [remark 60 on page 59](#)).
 - In a batch processing run, DesignManager continues to process subsequent amendment lines if possible (so that any subsequent errors in the attempted amendments can also be detected), but the amended version is not written to the source dataset (unless the ACCEPT amendment line is input: see [remark 59 on page 58](#)).
15. An amendment line can be:
- An insertion line or replacement line. These lines either have no initial keyword, but commence with a line number, or commence with the keyword LINE. See [remark 20 on page 49](#) through [remark 23 on page 50](#).
 - An editing instruction (DELETE, MOVE, COPY, CHANGE, FILL, BLANK). See [remark 24 on page 50](#) through [remark 48 on page 56](#).
 - A control instruction (DISPLAY, LOCATE, RENUMBER, ACCEPT, QUIT, END). See [remark 24 on page 50](#), [remark 25 on page 51](#), and [remark 49 on page 56](#) through [remark 61 on page 59](#).
16. Each amendment line must be wholly contained on one input line.
17. The first element of each amendment line must commence in the first character position of the input line.
18. Line numbers in amendment lines can have leading zeros (within the maximum length for line numbers of eight digits), or leading zeros can be omitted, at the user's option.
19. Amendments can be made in any order.
20. For an amendment line that commences with a line number, *m*:
- If source-line is not stated in the amendment line, DesignManager provides a default source-line consisting of space characters.
 - If the line number *m* does not already exist in the member's source record, source-line is inserted in the new version of the record, in the position, relative to other lines, determined by *m*.
 - If the line number *m* already exists in the member's source record, the line bearing that line number is replaced by source-line in the new version of the record.

21. An amendment line that commences with the keyword LINE operates as stated in [remark 20 on page 49](#), except that the line number for the insertion or replacement line, instead of being taken from the element *m* of the amendment line, is taken from an internal DesignManager memory set up by an earlier LOCATE amendment line (see [remark 52 on page 57](#) through [remark 56 on page 58](#)).
22. For an insertion line or a replacement line, the line to insert commences in the second character position following the line number or LINE keyword; that is, a single space separates the line number or LINE keyword from source-line. If additional spaces separate the line number or LINE keyword from the first element of source-line, these additional spaces form part of the line to insert, and are written into the source record. It is thus possible for users to ensure that any required alignment of elements in the source record is achieved.
23. It is not possible to amend the whole of a completely full source record input line by a replacement line (unless a different input device having a longer record length than that used for the original insertion of the line is employed). This is because the replacement line must include the line number or LINE keyword and the one space separator; so that it is not possible to specify new contents for the last *x* character positions of the line in the source record, where *x* is the number of digits coded for the line number, plus one for the separator space, or is five if the LINE keyword is used (or three if the LINE keyword is fully truncated). After replacement of the line, its last *x* character positions are spaces. A FILL or a CHANGE amendment line can be used to amend the last *x* character positions of the line.
24. When either of these amendment lines process:
 - Editing instructions (see [remark 26 on page 51](#) through [remark 48 on page 56](#))
 - DISPLAY, LOCATE, or RENUMBER control instructions (see [remark 49 on page 56](#) through [remark 58 on page 58](#))

Then these rules apply:

- If *m* is stated without THRU *n* (or, for a DELETE instruction, without TO *n*), the line numbered *m* must be present in the copy source record; if it is not present, the amendment line rejects and this error message is output:

```
m NOT FOUND - AMENDMENT IGNORED
```

(This is not relevant to a RENUMBER control instruction.)
- If *m* THRU *n* is stated (or, for a DELETE instruction, if *m* TO *n* is stated), all lines in the copy source record whose line numbers are in the range *m* to *n* are processed. The line numbers *m* and *n* need not exist in the record.
- If *m* THRU END is stated, all lines in the copy source record from the line numbered *m* (or if there is no line numbered *m*, from the line with the next higher line number), to the last line of the record, are processed. The line number *m* need not exist in the record. (This is relevant only to a DISPLAY or LOCATE control instruction.)
- If *x* is stated, and a line numbered *x* does not exist in the copy source record, the amendment line processes as though a line numbered *x* existed. (This is relevant only to a MOVE or COPY instruction.)

25. When these amendment lines process:
- Editing instructions (see [remark 26](#) through [remark 48 on page 56](#)).
 - DISPLAY control instructions (see [remark 49 on page 56](#) through [remark 51 on page 57](#)).

Then if LINE is stated, the number of the line to process is obtained from an internal DesignManager memory set up by an earlier LOCATE amendment line (see [remark 52 on page 57](#) through [remark 56 on page 58](#)).

26. A DELETE amendment line deletes the specified line or lines from the copy source record.

27. The amendment line:

$$\text{MOVE } \left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\} \text{ BEFORE } \left\{ \begin{array}{l} r \\ \text{LINE} \end{array} \right\}$$

inserts a copy of the line or lines specified by m [THRU n] or LINE immediately before the line specified by r or LINE. If THRU n is stated, the number of the line specified by r or LINE must not be in the range m to n . The inserted (moved) lines are numbered as stated in [remark 33 on page 52](#). The moved lines are deleted from their original position in the record.

28. The amendment line:

$$\text{MOVE } \left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\} \text{ TO } \left\{ \begin{array}{l} r \\ \text{LINE} \end{array} \right\}$$

inserts a copy of the line or lines specified by m [THRU n] or LINE in place of the line specified by r or LINE. If THRU n is stated, the number of the line specified by r or LINE must not be in the range m to n . The inserted (moved) lines are numbered as stated in [remark 34 on page 53](#). The moved lines are deleted from their original position in the record.

29. The amendment line:

$$\text{MOVE } \left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\} \text{ AFTER } \left\{ \begin{array}{l} r \\ \text{LINE} \end{array} \right\}$$

inserts a copy of the line or lines specified by m [THRU n] or LINE immediately after the line specified by r or LINE. If THRU n is stated, the number of the line specified by r or LINE must not be in the range m to n . The inserted (moved) lines are numbered as stated in [remark 35 on page 53](#). The moved lines are deleted from their original position in the record.

30. The maximum number of lines that a MOVE amendment line or a COPY amendment line can insert is 32,767.

31. A COPY amendment line operates in the same way as the corresponding MOVE amendment line (see [remark 27](#) through [remark 30](#)) except that:

- Copied lines are not deleted from their original positions in the record. (They could be subjected to renumbering: see [remark 33 on page 52](#) through [remark 35 on page 53](#).)
- The restriction on the MOVE amendment line, that if a THRU clause is present, the number of the line specified by r or LINE must not be in the range m to n , does not apply to the COPY amendment line.

32. In [remark 33](#) through [remark 35](#) below, this notation is used:
- p is the increment applied to line numbers when moved or copied lines are numbered, and when following lines have to be renumbered as a result of inserting moved or copied lines.
 - q is the line number of the line preceding line r .
 - r is a line number specified in, or determined by a BEFORE LINE, TO LINE, or AFTER LINE clause in a MOVE or COPY amendment line (see "[MODIFY Format](#)" on page 45).
 - s is the line number of the line following line r .
 - t (whose value cannot exceed 32,767) is the number of lines inserted by a MOVE or COPY amendment line.

33. The lines inserted by a:

$$\left\{ \begin{array}{l} \text{MOVE} \\ \text{COPY} \end{array} \right\} \left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\} \text{ BEFORE } \left\{ \begin{array}{l} r \\ \text{LINE} \end{array} \right\}$$

amendment line are numbered commencing at $(q + p)$ with increments of p .

where p is:

- The integer specified in or defaulted in the RENUMBER clause, if the RENUMBER clause is present in the command and if t (integer) is less than $(r - q)$; else
- 100, if $100t$ is less than $(r - q)$; else
- 10, if $10t$ is less than $(r - q)$; else
- 5, if $5t$ is less than $(r - q)$; else
- 2, if $2t$ is less than $(r - q)$; else
- 1, if t is less than $(r - q)$

If t is equal to or greater than $(r - q)$, then line r must be renumbered. In this case, the inserted lines together with line r are numbered commencing at r with increments of p , where p is:

- The integer specified in or defaulted in the RENUMBER clause, if the RENUMBER clause is present in the command and if t (integer) is less than $(s - r)$; else
- 100, if $100t$ is less than $(s - r)$; else
- 10, if $10t$ is less than $(s - r)$; else
- 5, if $5t$ is less than $(s - r)$; else
- 2, if $2t$ is less than $(s - r)$; else
- 1, if t is less than $(s - r)$

If t is equal to or greater than $(r - q)$ and is equal to or greater than $(s - r)$, then line s (and possibly the following lines) must also be renumbered. In this case, the inserted lines together with lines r and s and all following lines up to a point where there is a gap in the line numbers sufficiently large to accept the new line numbers of all the renumbered lines, or to the end of the record if no such gap exists, are numbered commencing at the lowest multiple of p that is greater than q , with increments of p , where p is:

- The integer specified in or defaulted in the RENUMBER clause, if the RENUMBER clause is present in the command; else
- 100

34. The lines inserted by a:

$$\left\{ \begin{array}{l} \text{MOVE} \\ \text{COPY} \end{array} \right\} \left\{ \begin{array}{l} m \text{ [THRU } n] \\ \end{array} \right\} \text{ TO } \left\{ \begin{array}{l} r \\ \text{LINE} \end{array} \right\}$$

amendment line are numbered commencing at r with increments of p .

where p is:

- The integer specified in or defaulted in the RENUMBER clause if the RENUMBER clause is present in the command and if t (integer) is less than $(s - r + 1)$;
- 100, if $100t$ is less than $(s - r + 1)$; else
- 10, if $10t$ is less than $(s - r + 1)$; else
- 5, if $5t$ is less than $(s - r + 1)$; else
- 2, if $2t$ is less than $(s - r + 1)$; else
- 1, if t is less than $(s - r + 1)$

If t is equal to or greater than $(s - r + 1)$, then line s and possibly the following lines, must also be renumbered. In this case, the inserted lines together with line s and all following lines up to a point where there is a gap in the line numbers sufficiently large to accept the line numbers of all the renumbered lines, or to the end of the record if no such gap exists, are numbered commencing at r , then at the lowest multiple of p that is greater than r , and thereafter with increments of p .

where p is:

- The integer specified in or defaulted in the RENUMBER clause, if the RENUMBER clause is present in the command; else
- 100

35. The lines inserted by an amendment line are numbered commencing at $(r + q)$ with increments of p :

$$\left\{ \begin{array}{l} \text{MOVE} \\ \text{COPY} \end{array} \right\} \left\{ \begin{array}{l} m \text{ [THRU } n] \\ \end{array} \right\} \text{ AFTER } \left\{ \begin{array}{l} r \\ \text{LINE} \end{array} \right\}$$

where p is:

- The integer specified in or defaulted in the RENUMBER clause if the RENUMBER clause is present in the command and if t (integer) is less than $(s - r)$; else

- 100, if $100t$ is less than $(s - r)$; else
- 10, if $10t$ is less than $(s - r)$; else
- 5, if $5t$ is less than $(s - r)$; else
- 2, if $2t$ is less than $(s - r)$; else
- 1, if t is less than $(s - r)$

If t is equal to or greater than $(s - r)$ then line s (and possibly the following lines) must also be renumbered. In this case, the inserted lines together with line s and all following lines up to a point where there is a gap in the line numbers sufficiently large to accept the new line numbers of all the renumbered lines, or to the end of the record if no such gap exists, are numbered commencing at the lowest multiple of p that is greater than r , with increments of p .

where p is:

- The integer specified in or defaulted in the RENUMBER clause, if the RENUMBER clause is present in the command; else
- 100

36. This amendment line scans the specified line or lines for occurrences of the specified string:

```
CHANGE { m [THRU n] } [ALL] 'string' [TO] 'string-2'
        { LINE }
```

If ALL is stated, all occurrences detected are replaced by *string-2*. If ALL is omitted, then on each line on which an occurrence of the specified string is detected, the first occurrence of the string is replaced by *string-2*; any subsequent occurrences of the string within the line are not replaced. These rules apply:

- The specified string, to be detected, must be wholly contained in a source record line; if it is split over two source record lines, it is not detected.
- If a source record line contains two or more delimited character strings (see "[Rules Governing Variables](#)" on page 19) and an occurrence of the characters to change crosses from one delimited string to the next, that occurrence is not detected unless the specified string includes the delimiters together with any spaces that may separate them in the source record line. That is, each source record line processes as a string of characters; concatenation of delimited strings that can occur in other processing contexts does not occur in amendment line processing.
- If *string-2* is of such a length that its substitution for an occurrence of string would cause a source record line to contain more than 254 characters, the source record line is left unchanged. (Line numbers are not included in the count of the characters in a source record line.) This error message is output:

```
CHANGE WOULD CAUSE line-number TO EXCEED MAXIMUM LENGTH
```

37. This amendment line scans all lines of the source record for occurrences of the specified string:

```
CHANGE [ALL] 'string' [TO] 'string-2'
```

If ALL is stated, all occurrences detected are replaced by *string-2*. If ALL is omitted, then on each line on which an occurrence of the specified string is detected, the first occurrence of the string is replaced by *string-2*; any subsequent occurrences of the string within the line are not replaced. The rules stated in [remark 36 on page 54](#) apply.

38. This amendment line replaces the specified line or each of the specified lines by a line containing the specified string, padded to the right with spaces.

```
FILL { m [THRU n] } 'string'  
      { LINE }
```

39. This amendment line replaces character position *a* of the specified line or of each of the specified lines by the specified string. In this case, *string* must be a single character:

```
FILL { m [THRU n] } COLUMNS a 'string'  
      { LINE }
```

40. This amendment line replaces character positions *a* to *b* of the specified line or of each of the specified lines by the specified string:

```
FILL { m [THRU n] } COLUMNS a thru b 'string'  
      { LINE }
```

If there are fewer characters in the string than in character positions *a* to *b*, the excess character positions are space filled to character position *b*. If there are more characters in the string than there are in character positions *a* to *b*, the amendment line rejects and an error message is output:

```
string INVALID CHARACTER STRING
```

41. This amendment line replaces character position *a* by the specified string, in every line of the source record:

```
FILL COLUMNS a 'string'
```

In this case, *string* must be a single character.

42. This amendment line replaces character positions *a* to *b* of every line of the source record by the specified string:

```
FILL COLUMNS a THRU b 'string'
```

If there are fewer characters in the string than there are in character positions *a* to *b*, the excess character positions are space filled to character position *b*. If there are more characters in the string than there are in character positions *a* to *b*, the amendment line rejects and an error message is output:

```
string INVALID CHARACTER STRING
```

43. This amendment line replaces the specified line or each of the specified lines by a line containing only space characters:

BLANK $\left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\}$

44. This amendment line replaces the character position *a* of the specified line, or of each of the specified lines by a space character:

BLANK $\left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\}$ COLUMNS *a*

45. This amendment line replaces character positions *a* to *b* of the specified line or of each of the specified lines by space characters:

BLANK $\left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\}$ COLUMNS *a* THRU *b*

46. This amendment line replaces character position *a* by a space character in every line of the source record:

BLANK COLUMNS *a*

47. This amendment line replaces character positions *a* to *b* of every line of the source record by space characters:

BLANK COLUMNS *a* THRU *b*

48. If the keyword PRINT is included in a CHANGE, FILL, or BLANK amendment line, all affected lines are output on the primary output device from the working storage copy source record.

49. This amendment line outputs on the primary output device the whole of the working storage copy source record:

DISPLAY

50. This amendment line outputs on the primary output device the specified line or lines from the working storage copy source record:

BLANK $\left\{ \begin{array}{l} m \text{ [THRU } n] \\ \text{LINE} \end{array} \right\}$ $\left\{ \begin{array}{l} \\ \text{END} \end{array} \right\}$

51. The DISPLAY amendment line enables the effects of earlier processed amendment lines to be examined at any stage of the processing of the command. In an interactive processing run, it provides an opportunity to identify mistakes and correct them immediately. Or, in conjunction with QUIT (see [remark 60 on page 59](#)) to abandon the amendment if a desired result is not achieved. In a batch processing run, a DISPLAY amendment line followed by a QUIT enables the effects of a complex series of amendment lines to be verified before the source dataset is updated; if the desired result is achieved by the amendment lines, the command can be repeated in a subsequent run without the DISPLAY and QUIT amendment lines.

52. This amendment line scans the whole of the copy source record:

```
LOCATE 'string'
```

Or, this amendment line scans the specified line or lines of the copy source record, for an occurrence of the specified string:

```
LOCATE m [THRU {n } ] 'string'
           {END }
```

The first line on which the string is detected is output on the primary output device; lines following the line that contained the first detected occurrence are not scanned. The specified string, and the line number of the line that contains the first detected occurrence of that string, are retained in internal DesignManager memories (the string memory and the line number memory respectively), until the processing of the MODIFY command terminates, or until the next amendment line processes:

```
LOCATE { [m [THRU {n } ]] 'string'
         {NEXT ['string']
         {ALL 'string' }
```

See further in [remark 53](#) through [remark 57 on page 58](#). If no occurrence of the string is located, an error message is output:

```
string NOT FOUND
```

The first two rules stated in [remark 36 on page 54](#) apply.

53. This amendment line is accepted only if a string has been located by a previous LOCATE instruction:

```
LOCATE NEXT ['string']
```

If there was no previous LOCATE instruction, or if the previous LOCATE instruction was unsuccessful, an error message is output:

```
NO PREVIOUS STRING GIVEN
```

54. This amendment line locates the next line (commencing at the line following the last located line) on which the last located string was detected, outputs the line on the primary output device, and updates the line number memory:

```
LOCATE NEXT
```

55. This amendment line operates in the same way as LOCATE 'string' except that it scans the copy source record, commencing at the line following the last located line, for an occurrence of the newly specified string (which replaces the previous string in the string memory):

```
LOCATE NEXT 'string'
```

56. This amendment line scans the whole of the copy source record, and outputs on the primary output device every line on which the specified string is detected:

```
LOCATE ALL 'string'
```

The first two of the rules stated in [remark 36 on page 54](#) apply. The string memory and the line number memory are both left empty by this control instruction.

57. The line number memory set up by a LOCATE control instruction is updated in these circumstances:
- If the line whose number is held in the memory is renumbered by a RENUMBER editing instruction (see [remark 58](#)).
 - If the line whose number is held in the memory is renumbered as a result of a MOVE or COPY editing instruction, as described in [remark 33 on page 52](#), [remark 34 on page 53](#), or [remark 35 on page 53](#); but if the line's number is altered because the line is itself MOVED, then the line number memory is left empty.

58. This amendment line renumbers all lines that have line numbers in the range m to n in the copy source record:

```
RENUMBER  $m$  THRU  $n$  [FROM  $v$ ] IN  $i$ 
```

If the FROM clause is present, the first line renumbered is renumbered as v ; if the FROM clause is absent, a default of FROM m is assumed, and the first line renumbered is renumbered accordingly. Subsequent lines in the specified range are renumbered in increments of i . DesignManager checks that v , if present, is greater than the line number of the line preceding the first line in the specified range, and that the new line number of the last renumbered line is less than the line number of the next following line, if any. If either of these conditions is not satisfied, the lines are not renumbered, and an error message outputs:

```
{  $v$  } INVALID INTEGER  
{  $i$  }
```

59. This amendment line, if included at the end of the amendment lines, instructs DesignManager to write the amended version of member-name to the source dataset, irrespective of whether any errors have been detected:

```
ACCEPT
```

Users should exercise great care if they utilize this control instruction, because its use could result in a corrupted source member. The next input line should be a DesignManager command. (The ACCEPT amendment line acts also as an amendment terminator.)

60. This amendment line prevents the writing of the amended version of the record to the source dataset:

QUIT

DesignManager then outputs this message:

AMENDMENT ABORTED

The following input line should be a DesignManager command. (The QUIT amendment line acts also as an amendment terminator.)

61. This amendment line has the same effect as ACCEPT in an interactive processing run, or has the same effect as an amendment terminator in a batch processing run:

END

The next input line should be a DesignManager dictionary mode or mode switch command. (The END amendment line acts also as an amendment terminator.)

62. If the amended version of the source record is not written to the source dataset (see [remark 13 on page 48](#), [Remark 14 on page 49](#), and [remark 60 on page 59](#)), the existing source record is retained, and no encoding takes place.

63. The automatic error recovery system treats MODIFY as an updating command. If a failure occurs during the updating of the source dataset, the automatic error recovery system restores the modeling dictionary to the state it was in immediately before the MODIFY command was actioned. If a failure occurs during the encoding phase of the command, that is, after this error message is output:

ALTERATION HAS CREATED NEW SOURCE RECORD

Then the automatic error recovery system restores the modeling dictionary to the state it was in at the time that the message was output. In the latter case the user should issue a further MODIFY command in a subsequent run.

MODIFY Examples

These examples illustrate MODIFY commands with their associated amendment lines:

```
MODIFY DATE ;  
00300 CATALOGUE 'GENERAL'  
;
```

```
MODIFY PERSONNEL RENUMBER 50;  
700 'PERSONNEL DEPARTMENT.'  
DELETE 800  
;
```

This example illustrates the use of editing instruction and control instruction amendment lines:

```
MODIFY PERSONNEL;  
CHANGE 300 THRU 500 ALL 'EMPLOYEE' TO 'EMP'  
DISPLAY 100 THRU 7000  
FILL 600 COLUMNS 1 THRU 5 "*****"  
DISPLAY 600 THRU 7000  
QUIT
```

This example illustrates a technique for inserting a line where the line numbers of the member are not known. If a string on the preceding or following line can be uniquely identified, LOCATE and LINE can be used to position the inserted text. (Alternatively, a print of the member could be obtained and the required line number could be found by inspection.):

```
MODIFY P3U02;  
LOCATE 'TELEPHONE CONTACT'  
190000 'OR IF NIGHT SHIFT: EXT. 210'  
MOVE 190000 AFTER LINE  
;
```

PRINT

The PRINT command is a modeling dictionary documentation command. The PRINT command outputs the source records of selected members of the modeling dictionary, or outputs selected parts of the source record of a member of the modeling dictionary.

PRINT Format

```
PRINT  
{ member-name [, member-name] ... } { ; }  
{ member-name LINES { integer } { . }  
{ line-number TO { line-number-2 } } }  
{ END } }
```

where:

member-name is the name of a member of the modeling dictionary.

integer is an unsigned integer of not more than 7-digits, being the number of lines from member-name's source record that are to be output.

line-number is an unsigned integer of not more than 7-digits.

line-number-2 is an unsigned integer, equal to or greater than line-number, of not more than 7-digits.

PRINT Remarks

1. If the LINES keyword is present in the command, only one member-name must be stated. If the LINES keyword is omitted from the command, one or more member-names can be stated. If two or more member-names are listed in the command, each except the last in the list must be followed by a comma; entries in the list may additionally be separated by spaces.
2. If member-name has no source record, processing continues with the next member-name or command and an error message is output:

```
member-name NO SOURCE PRESENT
```

3. If member-name has a source record, and the LINES keyword is not present in the command, the whole of member-name's source record is output on the primary output device.
4. If member-name has a source record, and the LINES keyword is present in the command and is followed by integer, the specified number of lines from the member's source record are output on the primary output device, commencing with the first line of the record.
5. If member-name has a source record, and the LINES keyword is present in the command and is followed by this clause:

```
line-number TO { line-number-2 }  
                { END }
```

Then lines from the member's source record are output on the primary output device, as determined by that clause. The line numbers *line-number* and *line-number-2* do not have to exist in the source record; lines with line numbers in the specified range (inclusive) are output.

PRINT Examples

This first example would output the whole source record of the member DATE:

```
PRINT DATE ;
```

This second example would output the first ten lines of the source record of the member PERSONNEL:

```
PRINT PERSONNEL LINES 10 ;
```

REMOVE

The REMOVE command is a modeling dictionary manipulation command. The REMOVE command removes members from the modeling dictionary.

REMOVE Format

```
REMOVE member-name [, member-name] . . . { ; }  
                                           { . }
```

where *member-name* is the name of a member as recorded in the modeling dictionary.

REMOVE Remarks

1. If no other member refers to *member-name*, *member-name*'s records are deleted from the modeling dictionary. If *member-name* has a data entries record (encoded or dummy), then before it is deleted, all *used by* pointers to *member-name* are deleted from other member's records. If these deletions result in any dummy data entries records being left with no *used by* pointers, such dummy records are deleted and the index dataset is updated accordingly.

2. If any other encoded member (that is, any other member with a non-dummy data entries record) refers to *member-name*, an error message is output:

```
member-name REFERRED TO BY OTHER MEMBERS - CANNOT BE REMOVED
```

No deletion of *member-name*'s records takes place, and processing continues with the next *member-name* or command.

3. If no record for *member-name* exists in the modeling dictionary, this error message is output:

```
member-name NOT PRESENT ON DICTIONARY
```

Processing continues with the next *member-name* or command.

4. If two or more *member-names* are listed in the command, each except the last in the list must be followed by a comma. Entries in the list may additionally be separated by spaces.
5. REMOVE is treated as an updating command by the automatic error recovery system, which operates individually in respect of each *member-name* appearing in the command.

REMOVE Example

```
REMOVE CURRENCY-INDICATOR;
```

```
REMOVE TOTALS-1974, TOTALS-1975, TOTALS-1976;
```

Specifications of Secondary Keywords and Clauses

The keywords and clauses specified here are available for use in the dictionary mode LIST command and in some Controller's commands. They provide further selection capabilities, additional to the primary selection capabilities defined in the individual command specifications. This enables the user to confine the operation of the command to those index-names that satisfy stated conditions. An index-name is the identification for an entry in the modeling dictionary's index dataset; it can be a member name, an alias, or a catalogue classification.

Any of these categories of index-name may be excluded by the primary selection keywords or clauses present in the particular command.

The selection capabilities defined here fall into two groups:

- Name-related-selection clauses
- Time-and-user-related-selection clauses

Name-related-selection Clauses

Name-related-selection clauses restrict the operation of the command to those index-names that appear within a defined part of the index dataset, or whose names contain a stated character string or are of a specified length.

Name-related-selection Format

$$\left[\left[\left[\text{FROM} \left\{ \begin{array}{c} "a" \\ a \end{array} \right\} \right] \left[\text{TO} \left\{ \begin{array}{c} "b" \\ b \end{array} \right\} \right] \right] \right] \left[\text{ONLY} \left\{ \begin{array}{c} "a" \\ a \end{array} \right\} \right] \left[\text{WHEN } condition \left[\left[\text{AND} \right] \left[\text{OR} \right] \left[\text{OR WHEN} \right] \right] condition \right] \dots \right]$$

where:

- a } are character strings of not more than 79 characters, representing commencing
- b } characters of entries in the index dataset. If a and b are both present, the alphanumeric value of b must be equal to or greater than the alphanumeric value of a .

condition is this format:

$$\left\{ \begin{array}{l} \left[\text{ANY} \right] \left[\text{EQ} \right] \left\{ \begin{array}{c} "string" \\ string \end{array} \right\} \\ \left[\overline{p} \left[\text{TO } q \right] \right] \left[\text{=} \right] \\ \left[\text{END } [-q] \left[\text{TO } \text{END } [-p] \right] \right] \left[\text{NE} \right] \\ \text{LENGTH} \left[\text{EQ} \right] integer \\ \left[\text{=} \right] \\ \left[\text{NE} \right] \\ \left[\text{GT} \right] \\ \left[\text{>} \right] \\ \left[\text{GE} \right] \\ \left[\text{LT} \right] \\ \left[\text{<} \right] \\ \left[\text{LE} \right] \end{array} \right\}$$

where:

p } are integers, with the value of p less than the value of q .
 q }

string is a printable character of from one to 32 printable characters.

EQ or = means equal to
NE means not equal to
GT or > means greater than
LT or < means less than
LE means less than or equal to

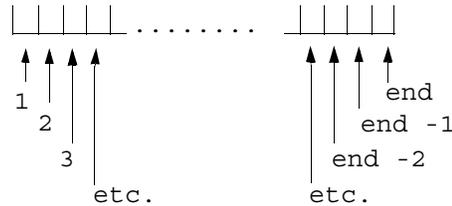
integer is an unsigned integer in the range 1 to 79, specifying a number of characters.

Name-related-selection Remarks

1. If a FROM clause is present in the relevant command, no index-names that appear in the index dataset before the position that is or would be occupied by index-names beginning with the character string defined in that clause are processed.
2. If a TO clause is present in a relevant command, no index-names that appear in the index dataset after the position that is or would be occupied by index-names beginning with the character string defined in that clause are processed.
3. If an ONLY clause is present in a relevant command, only those index-names that begin with the character string defined in that clause are processed.
4. If a WHEN clause is present in a relevant command, only those index-names that satisfy a condition or conditions specified in the WHEN clause are processed. Specifiable conditions are:
 - That stated character positions in the index-name should contain (specified by the element EQ or the element =) a string of characters that is the same as the character string stated immediately following the EQ or = element.
 - That stated character positions in the index-name should not contain (specified by the element NE) a string of characters that is the same as the character string stated immediately following the NE element.
 - That the length of the index-name should bear a stated relationship to a stated number of characters.

5. The character positions in index-names that are to be searched for the character string stated in a condition are specified thus:

- ANY specifies that the entire name is to be searched for an occurrence of the stated string.
- The character positions in an index-name are identified thus:



If, for example, a name were sixteen characters long, then its fourteenth character position could be identified by the integer 14, or by END -2.

- The integer *p* specifies that the character position identified by that integer is to be examined for an occurrence of the stated string. In this case, string must be a single character.
- *p* TO *q* specifies that part of the index-name commencing in the character position *p* and ending in character position *q* is to be examined for an occurrence of the stated string. In this case, string must not contain more than *q-p+1* characters.
- END specifies that the last character of index-name is to be examined for an occurrence of the stated string. In this case, string must be a single character.
- END -*q* specifies that the character position so identified is to be examined for an occurrence of the stated string. In this case string must be a single character.
- END -*q* TO END [-*p*] specifies that the part of the index-name commencing in the character position END - *q* and ending in character position END [-*p*] is to be examined for an occurrence of the stated string. In this case, string must not contain more characters than the part of index-name specified for the examination.

An occurrence of the stated string in the stated character position(s) or anywhere in the stated range of character positions in an index-name determines:

- That the condition is satisfied if EQ or = is present in the condition.
- That the condition is not satisfied if NE is present in the condition.

6. If an index-name is shorter than the stated string, or if an index-name is so short as to not include all character positions specified in a condition, then:

- The condition is not satisfied if EQ or = is present in the condition.
- The condition is satisfied if NE is present in the condition.

7. The connectors AND and/or OR connect a set of conditions that are to be evaluated from left to right to determine whether an index-name is to be processed. The connector OR WHEN introduces another set of conditions that is to be evaluated separately from the preceding set. If any of the separately evaluated sets of conditions gives a *satisfied* result, the index-name processes. (See ["Name-related-selection Examples." on page 66.](#))

8. These character strings are, in respect of delimiters, subject to the rules that apply to member names, stated in ["Terminators" on page 18](#):

- *a* in the FROM clause
- *b* in the TO clause
- *a* in the ONLY clause
- *string* in the WHEN clause

The delimiters are optional unless any of the additional characters that are available in the extended character set for member names appear in the string, in which case the delimiters are required.

9. FROM, TO, and WHEN clauses can be in any order. ONLY and WHEN clauses can be in either order. The position of these clauses must be as defined in individual command specifications.

Name-related-selection Examples.

For examples of the evaluation of a set of conditions specified in a WHEN clause, consider a dictionary which contains these index-names:

- ACCOUNT
- DEPARTMENT
- PAYROLL
- PACKET

This selection is processed left to right:

```
WHEN 1 = 'A' AND END = 'T' OR 1 = 'P' AND END = 'L'
```

It is thus equivalent to:

```
((1 = 'A' AND END = 'T') OR 1 = 'P') AND END = 'L'  
= ((ACCOUNT) OR 1 = 'P') AND END = 'L'  
= (ACCOUNT, PAYROLL, PACKET) AND END = 'L'  
= PAYROLL
```

The selection:

```
WHEN 1 = 'A' AND END = 'T' OR WHEN 1 = 'P' AND END = 'L'
```

is equivalent to:

```
(1 = 'A' AND END = 'T') OR (1 = 'P' AND END = 'L')  
= ACCOUNT, PAYROLL
```

Time-and-user-related-selection Clauses

Time-and-user-related-selection clauses restrict the operation of the command to those index-names that have been processed in particular ways within the selected time periods and/or by stated users.

Time-and-user-related-selection Format

See [Figure 3](#) (simple selection) and [Figure 4](#) (complex selection) for the time-and-user-related-selection formats:

Figure 3. Simple time-and-user-related-selection

```

IF [NOT] {
  {
    INTRODUCED
    AMENDED
    VERIFIED
  }
}
{
  time-selection [BY { [USER] user-name } ]
  MASTER
}
BY { [USER] user-name } [time-selection]
  MASTER

```

Figure 4. Complex time-and-user-related-selection

```

IF [ ( ] ... [NOT] [ ( ] ... {
  {
    INTRODUCED
    AMENDED
    VERIFIED
  }
}
{
  time-selection [BY { [USER] user-name } ]
  MASTER
}
BY { [USER] user-name } [time-selection]
  MASTER
}
[AND [ ( ] ... [NOT] [ ( ] ... {
  {
    INTRODUCED
    AMENDED
    VERIFIED
  }
}
{
  time-selection [BY { [USER] user-name } ]
  MASTER
}
BY { [USER] user-name } [time-selection]
  MASTER
}
[ ) ] ... [ ) ] ... [ ) ] ...

```

where *time-selection* is:

$$\left\{ \begin{array}{l} \text{ON } \underline{date} \left[\left\{ \begin{array}{l} \underline{AT} \quad \underline{time} \\ \underline{BEFORE} \\ \underline{AFTER} \\ \underline{BETWEEN} \quad \underline{time} \quad \underline{AND} \quad \underline{time} \end{array} \right\} \right] \\ \\ \left\{ \begin{array}{l} \underline{BEFORE} \\ \underline{AFTER} \end{array} \right\} \quad \underline{date} \left[\left\{ \begin{array}{l} \underline{AT} \quad \underline{time} \\ \underline{BETWEEN} \quad \underline{time} \quad \underline{AND} \quad \underline{time} \end{array} \right\} \right] \\ \\ \underline{BETWEEN} \quad \underline{date} \left\{ \begin{array}{l} \left[\underline{AT} \quad \underline{time} \right] \quad \underline{AND} \quad \underline{date} \left[\underline{AT} \quad \underline{time} \right] \\ \underline{AND} \quad \underline{date} \left[\underline{BETWEEN} \quad \underline{time} \quad \underline{AND} \quad \underline{time} \right] \end{array} \right\} \end{array} \right\}$$

where:

date is in a format determined by the DCUST installation macro. If DCUST is not tailored by the Controller, then the default format applies; that is: *day month year*.

The variables for the *date* default format elements of *day month year* are these:

day is one or two numeric characters.

month is either one or two numeric characters, or one of these undelimited character strings of three characters (which may be truncated to the extent indicated):

JAN FEB MAR APR MAY JUN
JUL AUG SEP OCT NOV DEC

year is two or four numeric characters.

The elements *day*, *month*, and *year* are separated by any non-alphanumeric printable character. A space character is regarded as a printable character. If a space character or any of the characters full stop, semicolon, comma, or right parenthesis is used as the separator character, then date must be enclosed in quotes. If the separator character is any other acceptable character, date may optionally be enclosed in quotes.

time is in a format determined by the DCUST installation macro. If DCUST is not tailored by the Controller, then the default format applies; that is: *hour minute second*.

The variables for the *time* default format elements of *hour minute second* are these:

hour is one or two numeric characters in the range 0 to 24

minute is one or two numeric characters in the range 0 to 59

second is one or two numeric characters in the range 0 to 59

The elements *hour*, *minute*, and *second* are separated by any non-alphanumeric printable character. A space character is regarded as a printable character. If a space character or any of the characters full stop, semicolon, comma, or right parenthesis is used as the separator character, then time must be enclosed in quotes. If the separator character is any other acceptable character, time may be optionally enclosed in quotes.

user-name is a character string of up to 32 printable characters, being the name of a user as registered in the data dictionary by the Controller.

Time-and-user-related-selection Remarks

Simple Selection

1. If IF INTRODUCED is stated, all index names that were introduced to the modeling dictionary as specified by the further subordinate clauses of this selection are processed. INTRODUCED means:
 - For a member, when its source record was first entered into the modeling dictionary; unless the member exists only as a dummy, in which case INTRODUCED means when its dummy data entries record was first created.
 - For an alias or catalogue classification, when its index record was first created.
2. If IF AMENDED is stated, all members whose source records were most recently amended as specified by the further subordinate clauses of this selection are processed.
3. If IF VERIFIED is stated, all members that were most recently successfully encoded (by any command that causes encoding) as specified by the further subordinate clauses of this selection are processed.
4. If IF NOT INTRODUCED is stated, all index-names that were not introduced to the modeling dictionary as specified by the further subordinate clauses of this selection are processed.
5. If IF NOT AMENDED is stated, all members whose source records have not been amended (by their most recent updating) as specified by the further subordinate clauses of this selection are processed.
6. If IF NOT VERIFIED is stated, all members whose most recent successful encoding was not as specified by the further subordinate clauses of this selection are processed.
7. IF [NOT] AMENDED and IF [NOT] VERIFIED are meaningless in respect of the primary selection categories ALIASES, CATALOGUES, or CATALOGS. If NOT is present they result in all aliases or catalogue classifications being processed; if NOT is absent, they result in no aliases or catalogue classifications being processed.
8. In time-selection, all times relate to the time at which the processing of index-name-by a command that INTRODUCED, AMENDED, or VERIFIED index-name started. The time records to the second.
9. In the time-selection clause:

ON date AT time

AT time-selection specifies a time that must match exactly the relevant time recorded for index-name. In other time-selection clauses, AT time specifies a limiting time of a period.

10. In the time-selection clauses:

 BETWEEN date AND date

 and:

 BETWEEN time AND time

 selection is inclusive of the two dates stated and two times stated. BEFORE or AFTER
 selection is exclusive of the date or time stated.
11. The keyword MASTER denotes operations performed by the Controller.

Complex Selection

12. [Remark 1 on page 69](#) through [remark 11](#) apply equally to complex selection.
13. Any number of simple selection conditions can be linked together into a condition set by AND or OR connectors.
14. Parentheses can be used to establish the order in which conditions are evaluated. If the number of open parentheses, "(", does not equal the number of close parentheses, ")", the command containing the time-and-user-related selection rejects. There is no restriction on the depth of nesting of parentheses.
15. If no parentheses are present, evaluation is from left to right.
16. One of the keywords INTRODUCED, AMENDED, or VERIFIED must be stated in the first condition of a condition set, but the keyword can be omitted from subsequent conditions. If none of the keywords is stated in a condition (other than the first condition) then the keyword that applied to the previous condition applies also to this condition.

Time-and-user-related-selection Examples

```
IF INTRODUCED AFTER '30 JUN 1981' BETWEEN '10.00.01'  
    AND '06.00.00'
```

```
IF AMENDED BY 'BROWN J'  
    BETWEEN '1 JUL 1981' AND '31 DEC 1981'
```

```
IF AMENDED BY 'SMITHSON' BEFORE '31 DEC 1981'  
    AND NOT VERIFIED BY 'HARRIS'  
    AFTER '23 FEB 1982'
```

Overview of the Design Mode Command

These two groups represent the type of design mode commands available in DesignManager:

- Workbench design area manipulation commands
- Workbench design area documentation commands

An overview of these two groups of commands is given in "[Workbench Design Area Manipulation Commands](#)" on page 71 and "[Workbench Design Area Documentation Commands](#)" on page 72 respectively. Any references to entities are applicable only for users who have the optional Enterprise Modeling facility (Selectable Unit DSR-EM10) installed. *Data-view* is the generic category indicating either a userview or an entity. In "[Design Mode Command Specifications](#)" on page 73, the command specifications are given in alphabetical sequence of command identifier: the notation used in format specifications is defined in "[Notation For Statement Formats](#)" on page ix. General rules that apply to coding of all commands are given in Chapter 2, "[DesignManager Language and Coding](#)," on page 15.

Workbench Design Area Manipulation Commands

Workbench design area manipulation commands have to do with the loading of design data from the modeling dictionary into the workbench design area, the generation of a third normal form (3NF) relational schema from the design data, and the storing and reloading of the contents of the workbench design area on and from an external device. The commands include:

- CLEAR
- MERGE
- DESIGN
- NAME
- ENDDSR
- FETCH
- STORE

Enter a **CLEAR command** at any time to remove all current data from the workbench design area. If, for example, it is determined on-line during the processing of a MERGE command that incorrect data is present in the workbench design area or that the data is derived from an inconsistent set of data-views, enter a CLEAR command to empty the workbench design area. Then, after correcting the data-views in the modeling dictionary, using the dictionary mode MODIFY command; see "[Dictionary Mode Command Specifications](#)" on page 28, re-enter the MERGE command for the corrected set of data-views.

The **MERGE command** cumulatively loads the functional and multivalued dependency information obtained from one or more encoded USERVIEW and/or ENTITY members into the workbench design area, and merges it with the existing content of the workbench design area. Unless the NO-VERIFY option is specified in the MERGE command, all data elements brought into the workbench design area (as part of the dependency data) must be *verified* ITEM or GROUP members (see ["MERGE" on page 79](#)) of the modeling dictionary. If dummy ITEMS are permissible, then NO-VERIFY must appear in the command. After a MERGE command processes, the resulting content of the workbench design area can:

- Display by means of SNAPSHOT, LIST, or REPORT commands (see ["Workbench Design Area Documentation Commands" on page 72](#)).
- Augment with dependency information from additional userviews via another MERGE command.
- Transform using the DESIGN command into a 3NF relational schema.

The **DESIGN command** initiates the DesignManager design procedure (see ["Generating a Third Normal Form Relational Schema" on page 132](#)) for generating a 3NF relational schema from the dependency data residing in the workbench design area. The procedure includes elimination of redundancy from the dependencies and formulation of 3NF relations in the workbench design area with designated keys. The generated relations can then be named, either interactively or in batch, using the NAME command. Reports on the generated schema and relations, and the dependency information is available via the design mode REPORT command.

The **STORE command** permits the contents of the workbench design area to be stored as a workbench file on an external device for further processing at a later time. Load workbench files into the workbench design area by using the FETCH command.

An **ENDDSR command** terminates a DesignManager run. The command is not necessary in batch mode.

Workbench Design Area Documentation Commands

These commands produce output from the workbench design area:

- SNAPSHOT
- LIST
- REPORT
- ECHO or NO-ECHO

Output commands may be used at any point during design mode processing to describe, in varying degrees of detail, the content of the workbench design area; for example:

- Unnormalized data contained after processing of a MERGE command and before a subsequent DESIGN command.
- Normalized data after successful execution of the DESIGN command, both before and after NAMEing of relations.

The **SNAPSHOT command** provides an overview of the contents of the workbench design area in terms of the number of data-views, userviews, entities, data elements, dependencies, and relations, and indicates the current state of the WBDA as empty or as containing either normalized or unnormalized data. A further message is output if the data is derived from inconsistent data-views (see ["MERGE" on page 79](#)).

The **LIST command** selectively lists the names of all dataviews, userviews, entities, data elements, dependencies, and/or relations currently held in the workbench design area. Options are provided for listing each category (except dependencies and relations) alphabetically, and for listing from each category (except relations) only the data that was brought into the workbench design area by the most recent MERGE command.

The **REPORT command** produces reports describing the content of the workbench design area. The REPORT command can produce seven categories of report:

- Data-view
- Userview
- Data Element Usage Analysis
- Logical Schema
- Intersecting Data Element
- Entity (if the Enterprise Modeling facility, selectable unit DSR-EM10, is installed)
- Load Factor Analysis (if the Load Factor Calculation facility, selectable unit DSR-PH10, is installed). The reports can be produced in summary or in detail.

The **ECHO command** causes all succeeding command statements to display before processing, unless a subsequent NO-ECHO command is entered. In batch, an ECHO command is assumed automatically at the beginning of every run.

Design Mode Command Specifications

This section describes the Design Mode Command specifications for the manipulation and documentation workbench design area commands.

CLEAR

The CLEAR command is a workbench design area manipulation command. Use the CLEAR command to remove all data from the workbench design area.

CLEAR Format

CLEAR { ; }
 { . }

CLEAR Remarks

At the beginning of a DesignManager run and until a valid MERGE command is accepted, the workbench design area contains no data; thus a CLEAR command is not required, but if input, it is accepted.

DESIGN

The DESIGN command is a workbench design area manipulation command. Use the DESIGN command to generate a third normal form (3NF) logical schema from a set of userviews and/or entities merged into the workbench design area.

DESIGN Format

```
DESIGN [UNCONDITIONALLY]
      [ [WITH] AUDIT [format-selection] ] { ; }
                                           { . }
```

where *format-selection* is:

```
{ DETAILS }
{ SUMMARY }
```

DESIGN Remarks

1. At least one valid MERGE command must precede the DESIGN command, with no intervening CLEAR or DESIGN command.
2. If the workbench design area is empty, the DESIGN command rejects and this message is output:

```
WORKBENCH DESIGN AREA EMPTY
DESIGN COMMAND REJECTED
```

Two or more dependencies must be present in the workbench design area for successful execution of a DESIGN command. If only a single dependency is present, the DESIGN command terminates and this message is output:

```
DESIGN COMMAND TERMINATED
DATA IGNORED UNTIL NEXT TERMINATOR OR END OF LINE
```

3. If, due to a previously entered DESIGN command, the workbench design area contains normalized data that has not been augmented by a subsequent MERGE command, the current DESIGN command rejects and this message is output:

```
WORKBENCH DESIGN AREA ALREADY CONTAINS NORMALIZED DATA
DESIGN COMMAND REJECTED
```

4. If the data in the workbench design area has been produced from inconsistent user views and/or entities (see the design mode ["MERGE" on page 79](#)) and the keyword UNCONDITIONALLY is not specified, the DESIGN command rejects and this message is output:

```
WORKBENCH DESIGN AREA PRODUCED FROM INCONSISTENT  
DATA-VIEWS  
DESIGN COMMAND REJECTED
```

If the data was not produced from inconsistent members, then UNCONDITIONALLY, if specified, is accepted but ignored.

5. The DESIGN command initiates the DesignManager design procedure (see ["The DesignManager Design Procedure" on page 138](#)) for the dependency data residing in the workbench design area. The procedure consists of two major functions:
 - Normalization of the dependencies
 - Generation of third normal form (3NF) relations
6. When processing of the DESIGN command is complete, the workbench design area contains one or more relations, each having been assigned a unique relation number. This number provides one method of assigning names to relations (see ["NAME" on page 82](#)), and is also used in the output produced by various design mode reporting commands. The generated relations constitute a 3NF relational schema which represents *all* the functional dependencies (FDs) and multivalued dependencies (MVDs) held in the workbench design area when the DESIGN command was entered, whether or not any FDs were altered or eliminated during the normalization process.
7. If the [WITH] AUDIT clause appears in the DESIGN command, a report (see [Chapter 5, "DesignManager Output," on page 109](#)) is output giving an *audit trail* of all the dependencies in the workbench design area which have been altered and/or removed in the first two steps of the design procedure (see ["The DesignManager Design Procedure" on page 138](#)).
8. The format-selection options are extended if the User Formatted Output facility (selectable unit DSR-UD30) is installed. See *ASG-DesignManager User Formatted Output* for details.

ECHO

The ECHO command is a workbench design area documentation command. The ECHO command causes each subsequent command statement, whether in dictionary or design mode (or the MODE command statement itself), to display on the output device before processing.

ECHO Format

```
ECHO { ; }  
      { . }
```

ECHO Remarks

1. The ECHO command remains in effect until a NO-ECHO command is input or the run terminates.
2. In batch, all commands normally display on the output device. An ECHO command is required only to override a previously entered NO-ECHO command. Commands entered on-line, however, will not display unless preceded by an ECHO command.

ENDDSR

The ENDDSR command is a workbench design area manipulation command. The ENDDSR command terminates a DesignManager run.

ENDDSR Format

ENDDSR { ; }
{ . }

ENDDSR Remarks

1. On acceptance of this command, a DesignManager run terminates. Under TSO it offers an alternative to /*.
2. The command is not necessary (but is accepted) when using DesignManager in batch.
3. ENDDSR may be input when DesignManager is in either dictionary mode or design mode.

ENDDSR Examples

```
ENDDSR .  
  
END ;
```

FETCH

The FETCH command is a workbench design area manipulation command. Use the FETCH command to load a previously STOREd workbench file held on an external device into the workbench design area.

FETCH Format

FETCH EXTERNAL-FILE { FILE1 } { ; }
{ FILE2 } { . }

where *FILE1* and *FILE2* are the names of two workbench files set up as part of the run-time procedure and held on an external device.

FETCH Remarks

1. The FETCH and STORE (see "[STORE](#)" on page 87) commands provide a quick mechanism for saving the contents of the workbench design area and reloading it for further operations at a later time. The commands avoid the necessity to re-MERGE (see "[MERGE](#)" on page 79) viewsets and data-views, if a design iteration is interrupted.
2. On receipt of a FETCH command, DesignManager CLEARS the workbench design area before loading the data in the specified workbench file. It is not possible therefore to combine the information in a workbench file with any other data held in the workbench design area or in the other external file. Once the workbench file is loaded, however, further information can be MERGED with the FETCHed workbench design area.
3. If the workbench file specified in the command does not hold a copy of a previously STOREd workbench design area, the workbench design area is not CLEARed, the command rejects with an inforamory message, and processing continues with the next command.
4. A workbench file specified in a FETCH command is unchanged by the command, and can be used in later FETCH commands.

LIST

The LIST command is a workbench design area documentation command. Use the LIST command to list selected categories of information in the workbench design area.

LIST Format

```
LIST [ALPHABETICALLY] [RECENT] selection { ; }  
{ . }
```

where *selection* is a list of one or more of these keywords:

- USERVIEWS
- ENTITIES
- DATA-VIEWS
- DATA-ELEMENTS
- DEPENDENCIES
- FDS
- MVDS
- RELATIONS

LIST Remarks

1. If selection includes more than one keyword, each except the last must be followed by a comma and (optionally) spaces. Each keyword specified causes a list to produce of all workbench design area content in the corresponding category, except as constrained by use of the keyword RECENT (see [remark 4 on page 78](#)). The output arranges in the order in which the keywords are specified in the command. (See also [remark 8 on page 78](#).)
2. If the keyword ALPHABETICALLY is present in the command, then each category specified lists in alphanumeric order, except as indicated in [remark 3 on page 78](#).
3. The ALPHABETICALLY keyword is present in a LIST DEPENDENCIES, LIST FDS, or LIST MVDS command, is accepted but has no effect. A LIST ALPHABETICALLY RELATIONS command produces a list of relations, in this order:
 - All NAMED relations (see ["NAME" on page 82](#)), if any, in alphanumeric order.
 - Followed by any remaining relations, in order of relation number.If ALPHABETICALLY is not present in a LIST RELATIONS command, the relations list in order of relation number.
4. If the keyword RECENT is specified in the command, then the action of the command restricts to that data which was brought into the workbench design area by the last MERGE command, except as indicated in [remark 5](#). A LIST RECENT USERVIEWS command, for example, results in a list of only those USERVIEWS currently in the workbench design area which were added by the most recent MERGE command.
5. The RECENT keyword, if present in a LIST RELATIONS command, is accepted but has no effect. (Relations are not MERGED into the workbench design area; they generate there by action of the DESIGN command.)
6. If there is no information in the workbench design area when the LIST command is input, a message outputs and the command rejects.
7. Similarly, if the keyword RELATIONS is selected and the workbench design area contains unnormalized data, the LIST command rejects in respect of that keyword and a message outputs; other keywords included in such a command are accepted.
8. This output produces when a particular keyword is selected, subject to the above remarks:
 - For USERVIEWS, data-view number and the name of each userview.
 - For ENTITIES, data-view number and the name of each entity.
 - For DATA-VIEWS, data-view number and the name of each userview and/or entity.
 - For DATA-ELEMENTS, data-element number and the name of each data element.
 - For FDS, for each elemental functional dependency (see ["MERGE" on page 79](#)) present, the data elements comprising the left-hand and right-hand sides, the dependency type, and the number of the dependency in the workbench design area (absolute dependency number).

- For MVDS, for each multivalued dependency present, the data elements comprising the left-hand and right-hand sides, the dependency type, and the number of the dependency in the workbench design area (absolute dependency number).
- For DEPENDENCIES, the information given for FDS and MVDS in order of absolute dependency number.
- For RELATIONS, for each relation present, the number, name (if one has been assigned with the NAME command), and the data element(s) comprising the primary key.

MERGE

The MERGE command is a workbench design area documentation command. The MERGE command merges information cumulatively from one or more encoded USERVIEW and/or ENTITY members of the modeling dictionary with the existing content of the workbench design area.

MERGE Format

```
MERGE [NO-VERIFY] MEMBERS member-name [, member-name] ... { ; }
                                     { . }
```

where *member-name* is the name of one of these:

- A verified USERVIEW
- A verified ENTITY
- A verified VIEWSET

MERGE Remarks

1. If two or more member names appear in the command, each except the last in the list must be followed by a comma; entries in the list may additionally be separated by spaces.
2. *Member-name* must be the name of a *verified* USERVIEW, ENTITY, or VIEWSET member of the modeling dictionary. It must be a USERVIEW, ENTITY, or VIEWSET member with an encoded data entries record whose source record has not been altered since the member was last encoded. Otherwise, *member-name* rejects for MERGEing, one of these error messages is output, and processing continues with the next named member or the next command:

```
member-name IS NOT AN EXISTING DICTIONARY MEMBER
```

```
MEMBER TYPE, member-type, OF member-name IS INVALID FOR MERGE  
COMMAND
```

```
MEMBER member-name HAS NO DATA ENTRIES RECORD
```

```
MEMBER member-name HAS A DUMMY DATA ENTRIES RECORD
```

```
MEMBER member-name HAS A SOURCE RECORD WHICH HAS BEEN ALTERED  
SINCE IT WAS LAST ENCODED
```

3. Unless NO-VERIFY appears in the command, each data element named must be a verified ITEM or GROUP member of the modeling dictionary (see [remark 2 on page 79](#)), such as either of these:

- In the data definition of member-name, if member-name is a USERVIEW or ENTITY.
- In the data definition of a USERVIEW or ENTITY directly or indirectly CONTAINED in member-name, if member-name is a VIEWSET.

Otherwise, the USERVIEW or ENTITY (in which the data element is named) rejects for MERGEing, one of these error messages is output for each unverified ITEM or GROUP named in the rejected member, and processing continues with the next named member or with the next command:

```
MEMBER data-element-name HAS A DUMMY DATA ENTRIES RECORD
```

```
MEMBER data-element-name HAS A SOURCE RECORD WHICH HAS BEEN  
ALTERED SINCE IT WAS LAST ENCODED
```

4. If NO-VERIFY appears in the command, the data elements described in [remark 3](#) need not be verified members of the modeling dictionary.
5. If, due to a previously entered DESIGN command, the existing workbench design area contains normalized data, then this data restores to its non-normalized state: any relations are removed (hence their names, if they have been NAMED, are lost), and any dependencies, as altered or removed during processing of the DESIGN command, are restored. Thus, the data in the workbench design area is just as it was after completion of the most recent MERGE command. Merging of the USERVIEW and/or ENTITY members specified in the current MERGE command can then begin.

For each USERVIEW and ENTITY member that either:

- Appears in the command as member-name
- Is directly or indirectly CONTAINED in member-name when member-name is a VIEWSET

the existing content of the workbench design area is augmented with a description of each functional dependency (FD) and multivalued dependency (MVD) appearing in (in the case of a USERVIEW) or represented by (in the case of an ENTITY) the member's clauses, except as indicated in [remark 6 on page 81](#). For a USERVIEW member, dependency information is read directly from its DEPENDENCIES clauses, whereas the content of an ENTITY member is first mapped (automatically) during the MERGE command, into an equivalent set of dependencies, as described in *ASG-DesignManager Enterprise Modeling*. As each dependency is added to the workbench design area, it is assigned two numbers:

- An absolute number indicating its position in the workbench design area.
- A relative number indicating the order in which it is added to the workbench design area from the member being processed.

These numbers are used in output reports produced by various other design mode commands.

6. Dependencies are added to the workbench design area as described in [remark 5 on page 80](#), except in these circumstances (tested for by DesignManager in the given order):
- Extraneous data elements are removed from the right-hand sides of dependencies, whether FDs or MVDs (see ["Merging Userviews into the Workbench Design Area" on page 130](#)).
 - FDs with composite right-hand sides (after of extraneous data elements) are not added workbench design area; they decompose into *elemental* FDs (see ["Merging Userviews into the Workbench Design Area" on page 130](#)) and then to the workbench design area.
 - Duplicates of dependencies already present in the workbench design area, are not added.
 - If an inconsistency occurs. That is, if an FD being processed has the same left-hand and right-hand side as a previously MERGED MVD, or vice versa, then the processing dependency is not added. A flag is set to indicate that an inconsistency exists in the USERVIEWS and/or ENTITYs being MERGED, this warning message is output, and processing continues with the next dependency or the next command:

```

ABOVE DEPENDENCY, DEFINED AS { FD }
                             { MVD }

IN member-type member-name,
WAS PREVIOUSLY DEFINED AS { MVD }
                           { FD }

```

where *member-type* is either USERVIEW or ENTITY. While the *inconsistency* flag is set, all subsequent DESIGN commands reject unless they contain the keyword UNCONDITIONALLY (see ["DESIGN" on page 74](#)). The *inconsistency* flag can be unset only by CLEARing the workbench design area (see ["CLEAR" on page 73](#)). Then the members can be corrected and reMERGED before entering a subsequent DESIGN command.

7. When all the USERVIEWS and/or ENTITYs specified directly or indirectly in the MERGE command have been merged into the workbench design area, any *implied* FDs holding amongst the data elements are added automatically to the workbench design area by the MERGE command. If one of the data elements in the workbench design area is a GROUP member of the modeling dictionary and another data element in the workbench design area is one of its CONTAINED members, then an FD is implied from the former to the latter and is added to the workbench design area (unless it already appears there). If, for example, the data element DATE is a GROUP member containing the ITEMS YEAR, MONTH, and DAY, and if, further, the data element MONTH appears in any dependency in the workbench design area, then, unless already present, this implied FD is added to the workbench design area:

```
DATE ---> MONTH
```

8. Member names must conform to the rules stated in ["Rules Governing Variables" on page 19](#). In particular, users who have the optional additional Enterprise Modeling facility (Selectable Unit DSR-EMI0) installed should observe the indicated constraints with respect to entity and data element names. Otherwise, naming conflicts may occur in the workbench design area (see the *ASG-DesignManager Enterprise Modeling*) during processing of a MERGE command.

NAME

The NAME command is a workbench design area manipulation command. The NAME command assigns names to one or more of the relations currently in the workbench design area.

NAME Format

```
NAME RELATIONS name-clause... { ; }  
                               { . }
```

where *name-clause* is:

```
{ K KEY data-element-name [, data-element-name] ... }  
{ N NUMBER relation-number }  
A AS relation-name
```

where:

data-element-name is the name of a data element in the workbench design area (and is held on the modeling dictionary as an ITEM or GROUP member).

relation-number is an unsigned integer used to identify a relation in the workbench design area (unique relation numbers are assigned to each relation during processing of the DESIGN command).

relation-name is a name which must be coded in accordance with the rules stated in ["Rules Governing Variables" on page 19](#) for coding modeling dictionary member names.

NAME Remarks

1. Use the NAME command to assign names to one or more relations generated by a preceding DESIGN command and currently contained in the workbench design area.
2. For each KEY clause appearing in the command, if the designated data element or set of data elements is a key of a relation currently contained in the workbench design area, the relation is assigned the name appearing in the associated AS clause, except as indicated in [remark 7 on page 83](#) and [remark 8 on page 83](#).
3. If any data-element-name specified in a KEY clause is not present in the workbench design area, the KEY clause and its associated AS clause are ignored, this error message is output, and processing continues with the next KEY or NUMBER clause or with the next command:

```
DATA ELEMENT data-element-name DOES NOT APPEAR ON THE WORKBENCH
```

4. If the data element or set of data elements designated in a KEY clause is not a key of any relation in the workbench design area; the KEY clause and its associated AS clause are ignored, this error message is output, and processing continues with the next KEY or NUMBER clause or with the next command:

```
NO RELATION EXISTS WITH THE GIVEN KEY
```

5. For each NUMBER clause appearing in the command, if the designated relation number is a positive integer less than or equal to the number of relations in the workbench design area, then the relation currently identified by relation-number is assigned the name appearing in the associated AS clause, except as indicated in [remark 7](#) and [remark 8](#).
6. If the relation number designated in a NUMBER clause either is a positive integer greater than the number of relations in the workbench design area or is not a positive integer, then the NUMBER clause and its associated AS clause are ignored, the appropriate one of these two error messages is output, and processing continues with the next KEY or NUMBER clause or with the next command:

```
RELATION NUMBER GIVEN GREATER THAN NUMBER OF  
RELATIONS
```

```
INVALID RELATION NUMBER relation-number SPECIFIED
```

7. Relation-name must conform to the rules stated in ["Rules Governing Variables" on page 19](#) for coding modeling dictionary member names. Otherwise, the AS clause in which relation-name appears is ignored along with the associated KEY or NUMBER clause which precedes it, this error message is output, and processing continues with the next KEY or NUMBER clause or with the next command:

```
INVALID RELATION NAME relation-name
```

8. If relation-name has already been assigned to a relation contained in the workbench design area, in consequence of a preceding KEY or NUMBER clause or a previously executed NAME command, the AS clause in which relation-name appears is ignored along with the associated KEY or NUMBER clause which precedes it, this error message is output, and processing continues with the next KEY or NUMBER clause or with the next command:

```
relation-name IS ALREADY USED AS THE NAME OF RELATION  
relation-number
```

9. If the relation identified by a KEY or NUMBER clause has already been assigned a name via a preceding KEY or NUMBER clause or a previously executed NAME command, the first name assigned is overwritten and replaced by the current name, and this message is output:

```
RELATION relation-number IS RENAMED relation-name-1  
REPLACING relation-name-2
```

NO-ECHO

The NO-ECHO command is a workbench design area documentation command. The NO-ECHO command suppresses the display of subsequent command statements on the output device.

NO-ECHO Format

```
NO-ECHO { ; }
        { . }
```

NO-ECHO Remarks

1. The NO-ECHO command suppresses display of all subsequent command statements on the output device until an ECHO is entered command or the run terminates.
2. Except when running in batch (see [remark 2 of "ECHO" on page 75](#)), commands normally do not display. A NO-ECHO command is required only to override a previously entered ECHO command.

REPORT

The REPORT command is a workbench design area documentation command. The REPORT command produces summary or detail reports on categories of data in the workbench design area.

REPORT Format

```
REPORT report-selection [format-selection] { ; }
                                           { . }
```

where *report-selection* is:

| | | | | | | |
|---|----------------------------|---|---|---------------------------|---|---------|
| { | DATA-ELEMENTS | } | { | ALL | } | [ALPHA] |
| | DATA-VIEWS | | | RECENT | | |
| | USERVIEWS | | | NAMES <i>name-list</i> | | |
| | ENTITIES | | | NUMBERS <i>range-list</i> | | |
| | LOAD-FACTORS | | | | | |
| { | LOGICAL-SCHEMA | } | { | ALL | } | [ALPHA] |
| | | | | NAMES <i>name-list</i> | | |
| | | | | NUMBERS <i>range-list</i> | | |
| { | INTERSECTING-DATA-ELEMENTS | } | } | | } | |

where:

name-list is a list of names, separated by commas, of data elements, userviews, entities, or relations. The list must contain names of one kind only, as indicated by the preceding report category keyword. Only data elements when the keyword is DATA-ELEMENTS or LOAD-FACTORS, only userviews when the keyword is USERVIEWS, only entities when the keyword is ENTITIES, and only relations when the keyword is LOGICAL-SCHEMA. The only exception to this rule occurs when the keyword is DATA-VIEWS, when name-list may contain the names of both userviews and entities.

range-list is a list of numeric ranges separated by commas. Each numeric range is expressed as *n1* [TO *n2*] where *n1* and *n2* are numbers assigned in the workbench design area to data elements, userviews, entities, or relations. The list must contain numbers of one kind only as indicated by the preceding report category keyword, except when the keyword is DATA-VIEWS. If present, *n2* must be greater than *n1*.

format-selection is:

$\left\{ \begin{array}{l} \underline{\text{SUMMARY}} \\ \underline{\text{DETAILS}} \end{array} \right\}$

REPORT Remarks

1. The reports produced by this command are described in [Chapter 5, "DesignManager Output," on page 109](#).
2. In these remarks, *report category* refers to the first mandatory keyword (for example DATA-ELEMENTS, USERVIEWS, or LOGICAL-SCHEMA) following the command identifier.
3. If the report category specified is LOGICAL-SCHEMA and the data in the workbench design area is unnormalized, that is, a DESIGN command is not executed or is followed by one or more MERGE commands, then the user is informed and the REPORT command rejects.
4. Some information that can be output by the REPORT command for report categories other than LOGICAL-SCHEMA is available only after the data in the workbench design area has been normalized. Otherwise, such information is not included in the report. Use a SNAPSHOT command to ascertain whether or not the workbench design area contains normalized data.
5. If there is no data in the workbench design area when the REPORT command issues, the user is informed, and no report is output.
6. If the keyword ALL is specified, then all of the data in the workbench design area, within the report category selected, is reported. The data is reported in order of workbench design area number unless ALPHA is also specified.

7. If the keyword RECENT is specified, then, for the report category selected, only data brought into the workbench design area by the last MERGE command will be reported. The data is reported in order of workbench design area number unless ALPHA is also specified. For instance, REPORT DATA-ELEMENTS RECENT reports only those data elements brought in by the last MERGE command. This facility is especially useful for incremental design techniques when an already existing workbench design area is augmented by a MERGE command. The RECENT option then allows the user to examine only the incremental data.
8. The keyword NAMES allows the user to select for reporting within a particular report category, only the data named in name-list. The data reports in the order listed unless ALPHA is also specified.
9. When the keyword NAMES is used for the LOGICAL-SCHEMA report category, it is used only to select relations which have been named (by the NAME command) for reporting.
10. The keyword NUMBERS allows the user to select for reporting, within the specified report category, only the data whose workbench design area number is specified in range-list. The data is reported in the order listed unless ALPHA is also specified. This is the only way to select a subset of relations in the LOGICAL-SCHEMA report category when these relations are as yet unnamed.
11. If the keyword ALPHA is specified, then, for the report category selected, the data reports in alphanumeric order. If the report category is LOGICAL-SCHEMA, then named relations, if any, report before any unnamed relations, the former in alphanumeric order and the latter in order of relation number.
12. If the report category is INTERSECTING-DATA-ELEMENTS, no further keywords (in report-selection) can be specified. The report produces for all intersecting data elements in the order in which they are encountered in the workbench design area.
13. If no format-selection is specified, the default selection is SUMMARY.

SNAPSHOT

The SNAPSHOT command is a workbench design area documentation command. The SNAPSHOT command produces a summary of the data in the workbench design area.

SNAPSHOT Format

SNAPSHOT { ; }
 { . }

SNAPSHOT Remarks

1. The SNAPSHOT command produces a summary report on the state of the workbench design area. It shows:
 - Whether the workbench design area is empty.
 - Whether or not the workbench data is normalized.
 - An indication, where necessary, that the workbench design area was produced from inconsistent data-views.
 - The total numbers of:
 - Data-views (userviews plus entities)
 - Userviews
 - Entities
 - Data-elements
 - Dependencies (including any redundant functional dependencies and/or inconsistent multivalued dependencies if the workbench data is normalized)
 - Relations
 - The utilization of the workbench design area expressed as a percentage of the total.

STORE

The STORE command is a workbench design area manipulation command. Use the STORE command to STORE on an external device a workbench file representing the contents of the workbench design area.

STORE Format

$$\text{STORE } \underline{\text{EXTERNAL-FILE}} \left\{ \begin{array}{l} \underline{\text{FILE1}} \\ \underline{\text{FILE2}} \end{array} \right\} \left\{ \begin{array}{l} ; \\ . \end{array} \right\}$$

where *FILE1* and *FILE2* are the names of two workbench files set up as part of the run-time procedure and held on an external device.

STORE Remarks

1. The STORE (and FETCH) commands provide a fast mechanism for saving the contents of the workbench design area and reloading it for further operations at a later time.
2. If the workbench file name (FILE1 or FILE2) specified in the command is not set up as part of the run time procedure, the command rejects and an informatory message is output.
3. The information STOREd in the workbench file includes all flags set by previous processing, for example, the normalization and consistency flags.
4. If the workbench file name specified in the command already contains data, that data is overwritten with the contents of the current workbench design area.
5. On completion of a successful STORE command, the content of the workbench design area is unchanged.

4

DesignManager Member Definition Statements

Overview of the Member Definition Statements

The *members* of a DesignManager modeling dictionary comprise data definitions.

Enter data definitions into the modeling dictionary from data definition statements, each must follow immediately after a dictionary mode ADD command, which initiates the process of entry. The MODIFY command can modify data definitions once they are in the modeling dictionary. (The ADD and MODIFY commands are defined in [Chapter 3, "DesignManager Commands," on page 23.](#))

A data definition statement is the means by which a user describes design data to DesignManager for insertion into a modeling dictionary. In DesignManager used alone, the term *member definition statement* is synonymous with *data definition statement*. (When DesignManager is integrated with DataManager, commands can also be stored in the dictionary, and the term *member definition statement* covers both data definition statements and stored command statements.)

General rules that apply to the coding of all member definition statements are given in [Chapter 2, "DesignManager Language and Coding," on page 15.](#)

The statement identifier at the beginning of a member definition statement is a member type identifier. The keyword is one of these:

- ITEM
- GROUP
- USERVIEW
- VIEWSET

In addition to the four basic member types, various DesignManager optional additional facilities provide facilities for defining further member types in the modeling dictionary. The definition statement specifications for these optional additional member types are given in the relevant DesignManager facility publications (see [Chapter 7, "Optional Additional Facilities," on page 165.](#))

The effects of integrating DesignManager with DataManager on the member types and their definitions are described in [Chapter 8, "DesignManager/DataManager Integration," on page 169.](#)

["Data Definition Statement Specifications" on page 91](#) of this chapter is concerned only with the data definition statements of the four basic DesignManager member types which are discussed and defined in [Chapter 1, "Introduction to DesignManager," on page 1](#).

Note: _____

In ["Data Definition Statement Specifications" on page 91](#), any references to ENTITY members, member names, and member types are applicable only for users who have the optional Enterprise Modeling facility (selectable unit DSR- EMI0) installed.

The terminator at the end of a member definition statement must be placed on a separate input line, in the first character position of the line, and must be followed by a space if a further DesignManager command is input on the same line.

A member definition statement consisting of a terminator alone can be regarded as an empty member definition statement in that it results in the generation of an empty source record for the member named in the command.

The format defined for the body of a member definition statement depends on the member type.

Data definition statement specifications for the different member types are given in ["Data Definition Statement Specifications" on page 91](#), in hierarchical order of member type, commencing with the ITEM data definition statement. Certain clauses that can be used in the member definition statements for any member type are defined in ["Common Clause Specifications" on page 99](#), so as to avoid excessive repetition and to simplify the presentation of the individual member type definition statement specifications. The notation used in format specifications is defined in ["Notation For Statement Formats" on page ix](#).

A member definition is inserted initially into a source record in the modeling dictionary. From this, it is subsequently encoded and the encoded representation is inserted into a data entries record. (The structure of the modeling dictionary is discussed further in [Chapter 1, "Introduction to DesignManager," on page 1](#).) Validation of the syntax of the member definition is not performed until the time of encoding, when the encoded representation rejects if any errors are detected.

A member definition contained in a source record differs from a member definition statement in two respects:

- The terminator is not included in the source record.
- In the source record, the input lines are numbered. The line numbers are determined by the command(s) by which the source record was created and/or amended.

Note: _____

The member's name does not form part of the member definition statement, but is declared in the command by which the member is entered into the modeling dictionary. From the command, the member-name is entered into the index dataset. Alternative names by which the member may be known can, however, be declared in the member definition statement (see the ["ALIAS Clause" on page 101](#)).

The variable `member-name` appearing in member definition statement specifications can alternatively be coded as `member-name` (that is, as a delimited character string). This alternative *must* be adopted if the `member-name` includes characters that are not permitted in the basic `member-name` character set (see ["Rules Governing Variables" on page 19](#)). To avoid further complicating statement specifications, the alternative way of declaring member names is not shown in the statement formats. Wherever the variable `member-name` appears in a member definition statement format, therefore, it should be interpreted as offering the choice:

```
{ member-name }  
{ 'member-name' }
```

The same applies where a subset of `member-name` is specified; for example, if `item-name` is specified, it can be coded as `item-name`.

Data Definition Statement Specifications

This section describes the four member type identifier keywords. Include one of these at the beginning of a member definition statement as a statement identifier.

ITEM Keyword

ITEM Format

```
ITEM  
  
[common clauses]  
  
{ ; }  
{ . }
```

where *common clauses* are any of these clauses, as defined in ["Common Clause Specifications" on page 99](#), in any order:

| | |
|-----------------------------|--------------------------------|
| <u>ACCESS-AUTHORITY</u> | <u>FREQUENCY</u> |
| <u>ADMINISTRATIVE -DATA</u> | <u>NOTE</u> |
| <u>ALIAS</u> | <u>OBSOLETE-DATE</u> |
| <u>CATALOGUE</u> | <u>QUERY</u> |
| <u>COMMENT</u> | <u>SECURITY-CLASSIFICATION</u> |
| <u>DESCRIPTION</u> | <u>SEE</u> |
| <u>EFFECTIVE-DATE</u> | |

ITEM Remarks

1. Common clauses can be present in any type of data definition statement; they are therefore defined separately in "[Common Clause Specifications](#)" on page 99. Not more than one of each of these clauses can be declared. If a common clause has a subordinate clause or keyword, the subordinate clause or keyword must not be truncated to an extent where it becomes ambiguous with any other clause identifier or other keyword available in the data definition syntax for this member type.
2. A record containing the item's data definition statement can be inserted into the modeling dictionary's source dataset by the dictionary mode ADD command (see [Chapter 3, "DesignManager Commands,"](#) on page 23). If no syntax errors are detected, an encoded record is subsequently generated and inserted into the data entries dataset.

ITEM Example

```
ITEM
ALIAS 'OFFICE-NUMBER'
CATALOGUE ADMIN, LOCATION
NOTE 'OFFICES ARE UNIQUELY IDENTIFIED BY'
      'OFFICE-NUMBER AND BUILDING-NUMBER'
;
```

GROUP Keyword

GROUP Format

```
GROUP
[CONTAINS content [,content]...]
[common clauses]
{ ; }
{ . }
```

where:

content is the name of an item or group.

common clauses are any of these clauses, as defined in "[Common Clause Specifications](#)" on page 99, in any order:

| | |
|-----------------------------|----------------------|
| <u>ACCESS-AUTHORITY</u> | <u>FREQUENCY</u> |
| <u>ADMINISTRATIVE -DATA</u> | <u>NOTE</u> |
| <u>ALIAS</u> | <u>OBSOLETE-DATE</u> |
| <u>CATALOGUE</u> | <u>QUERY</u> |

USERVIEW Keyword

USERVIEW Format

```

USERVIEW [ { RELATIVE FREQUENCY } n] [ RESPONSE-TIME m]
          { RF }
dependency clause...
[ common-clauses ]
{ ; }
{ . }
    
```

where:

n is an unsigned integer indicating the frequency of access to the userview being defined.

m is an unsigned integer indicating the response time required for data accessed by the process represented by the userview.

dependency-clause is:

```

DEPENDENCIES
{ LEFT-HAND SIDE } data-element-name
{ LHS } [, data-element-name] ...
right-hand-side-clause [right-hand-side-clause] ...
    
```

where:

data-element-name is the name of a data element that is either held in the modeling dictionary as an ITEM or GROUP member, or is entered in the modeling dictionary as a dummy ITEM.

right-hand-side-clause is:

```

{ { FUNCTIONAL DEPENDENCY } }
{ { FD } }
{ { MULTIVALUED-DEPENDENCY } } [k]
{ { MVD } }
{ RIGHT-HAND-SIDE } data-element-name
{ RHS } [, data-element-name] ...
    
```

where:

k is an unsigned integer, being the multiplicity for a multivalued dependency. It is the average number of values (or sets of values) determined for the right-hand side of the dependency by a given value of the left-hand side. If not specified in a dependency, the default value is taken to be one. The multiplicity of a functional dependency is always one.

common clauses are any of these clauses, as defined in "[Common Clause Specifications](#)" on page 99, in any order:

| | |
|-----------------------------|--------------------------------|
| <u>ACCESS-AUTHORITY</u> | <u>FREQUENCY</u> |
| <u>ADMINISTRATIVE -DATA</u> | <u>NOTE</u> |
| <u>ALIAS</u> | <u>OBSOLETE-DATE</u> |
| <u>CATALOGUE</u> | <u>QUERY</u> |
| <u>COMMENT</u> | <u>SECURITY-CLASSIFICATION</u> |
| <u>DESCRIPTION</u> | <u>SEE</u> |
| <u>EFFECTIVE-DATE</u> | |

USERVIEW Remarks

1. If the RELATIVE-FREQUENCY and/or RESPONSE-TIME clauses appear in the data definition statement, then they must precede all dependency-set clause specifications.

DesignManager users whose installation includes the Load Factor Calculation optional additional facility (selectable unit DSR-PH10) can use these clauses to input data for the calculation of load factors.

Any userview for which load factors are to be calculated should contain one of each of these clauses in its member definition statement. The value of *n* in the RELATIVE-FREQUENCY clause should be set to the relative frequency with which each data element that appears in the userview's dependencies is to be accessed by that userview. The value of *m* in the corresponding RESPONSE-TIME clause should be the time period within which each constituent data element must be accessed from that userview.

2. Each dependency-clause represents a list of functional dependencies (FDs) and/or multivalued dependencies (MVDs) all having the same left-hand side (see "[Structure and Definitions](#)" on page 124 and the "[Glossary](#)" on page 245). That is, the same set of data elements comprises the left-hand side of each dependency depicted in a dependency-clause. For example:

```
DEPENDENCIES
LHS   A, B, C  FD  RHS  D
          FD  RHS  E, F
          MVD  RHS  G
```

represents this list of dependencies:

```
A+B+C ---> D
A+B+C ---> E+F
A+B+C ---> G
```

3. The common clauses listed under ["USERVIEW Format" on page 94](#), can be present in any type of data definition statement: they are therefore defined separately (in ["Common Clause Specifications" on page 99](#)). Not more than one of each of these clauses can be declared. If a common clause has a subordinate clause or keyword, then the subordinate clause identifier or subordinate keyword must not be truncated to an extent where it becomes ambiguous with any other clause identifier or other keyword that is available in the data definition syntax for this member type.
4. Common clauses can be declared in any order in any position between the userview statement identifier and the terminator, except that they must not appear within a dependency-clause.
5. A record containing the userview's data definition statement can be inserted into the modeling dictionary's source dataset by the dictionary mode ADD command (see [Chapter 3, "DesignManager Commands," on page 23](#)). If no syntax errors are detected, then an encoded record subsequently generates and inserts into the data entries dataset.

If, when the encoded record is generated for the userview, any ITEM or GROUP member whose name appears in the userviews data definition statement has no data entries record, then a dummy data entries record is generated for that member containing *used-by* pointers (see ["Modeling Dictionary Records" on page 8](#)) to the userview being encoded. The dummy record is created as a dummy item record.

6. If, when the encoded record is generated for the userview, any data-element-name appearing in the userview's data definition statement is the name of an existing encoded or dummy member of type other than ITEM or GROUP, this error message is output and encoding of the userview is rejected (validation of the rest of the data definition statement continues):

```
Member-type member-name IS NOT A VALID MEMBER-TYPE IN THIS
CONTEXT
```

7. Users who have the Enterprise Modeling optional additional facility (selectable unit DSR-EM10) installed should observe the indicated constraint with respect to data element names.

USERVIEW Examples

```
USERVIEW
RELATIVE-FREQUENCY 5
RESPONSE-TIME 10
DEPENDENCIES
LEFT-HAND-SIDE EMP-NAME FUNCTIONAL-DEPENDENCY
                RIGHT-HAND-SIDE ADDRESS,
                SALARY
DEPENDENCIES
LHS EMP-CODE FD RHS EMP-NAME
;
```

VIEWSET Keyword**VIEWSET Format**

```

VIEWSET
[CONTAINS member-name [,member-name] ...]
[common clauses]
{ ; }
{ . }

```

where:

member-name is the name of a USERVIEW, ENTITY, or VIEWSET.

common clauses are any of these clauses, as defined in "[Common Clause Specifications](#)" on page 99, in any order:

| | |
|-----------------------------|--------------------------------|
| <u>ACCESS-AUTHORITY</u> | <u>FREQUENCY</u> |
| <u>ADMINISTRATIVE -DATA</u> | <u>NOTE</u> |
| <u>ALIAS</u> | <u>OBSOLETE-DATE</u> |
| <u>CATALOGUE</u> | <u>QUERY</u> |
| <u>COMMENT</u> | <u>SECURITY-CLASSIFICATION</u> |
| <u>DESCRIPTION</u> | <u>SEE</u> |
| <u>EFFECTIVE-DATE</u> | |

VIEWSET Remarks

1. The VIEWSET member is a grouping of USERVIEW and/or ENTITY members each specified either directly in the CONTAINS clause or indirectly via a VIEWSET named (as a subset of the viewset being entered) in the CONTAINS clause.
2. Member names must always conform to the coding rules stated in "[Rules Governing Variables](#)" on page 19. In particular, users who have the optional Enterprise Modeling facility (selectable unit DSR-EMI0) installed should observe the indicated constraints with respect to entity and data element names.
3. If more than one member name is declared in the CONTAINS clause, each except the first must be preceded by a comma and optionally by spaces.

4. The VIEWSET member can be used to group userviews and/or entities into logically related sets (for example, all userviews describing a particular application) and also provides a convenient method for merging a large number of userviews and/or entities from the modeling dictionary into the workbench design area. For example, instead of entering a MERGE command (see "[Design Mode Command Specifications](#)" on page 73) naming each userview and each entity separately, the user can obtain the same result by entering a simple MERGE command in which only a single viewset is specified.
5. Common clauses, listed under "[VIEWSET Format](#)" on page 97, can be present in any type of member definition statement; they are therefore defined separately, in "[Common Clause Specifications](#)" on page 99. Not more than one of each of these clauses can be declared. If a common clause has a subordinate clause identifier or keyword, the subordinate clause identifier or subordinate keyword must not be truncated to an extent where it becomes ambiguous with any other clause identifier or other keyword available in the data definition syntax for this member type.
6. Common clauses can be declared in any order in any position between the VIEWSET statement identifier and the terminator, except that they must not appear within a CONTAINS clause.
7. A record containing the viewset's data definition statement can be inserted in the modeling dictionary's source dataset by the dictionary mode ADD command (see "[Primary Command Specifications](#)" on page 28). If no syntax errors are detected, an encoded record is subsequently generated and inserted into the data entries dataset.

If, when the encoded record is generated for the viewset, any member whose name appears in the viewset's data definition statement has no data entries record, a dummy data entries record is created for that member containing *used-by* pointers (see "[Modeling Dictionary Records](#)" on page 8) to the viewset being encoded. The dummy record is created as a dummy viewset record.

If a member named in the viewset's data definition statement is an existing dummy member of type ENTITY (see *ASG-DesignManager Enterprise Modeling*) or VIEWSET, the member type of the dummy is not changed when the viewset is encoded.

8. If, when the encoded record is generated for the viewset. Any member name appearing in the viewset's data definition statement is the name of an existing encoded or dummy member of type other than USERVIEW, ENTITY, or VIEWSET, encoding of the viewset is rejected. This error message is output and validation of the remainder of the data definition statement continues:

```
member-type member-name IS NOT A VALID  
MEMBER-TYPE IN THIS CONTEXT
```

VIEWSET Examples

```
VIEWSET  
CONTAINS EMPLOYEE, DEPARTMENT, PROJECT  
;
```

Common Clause Specifications

Common clauses are clauses that can occur in any member definition statement; that is, they are common to all member types. To avoid excessive repetition in the DesignManager documentation, therefore, they are defined separately in this section, to which reference is made from the various data definition statement specifications. The extent to which the clause identifier keyword can be truncated depends on the type of member definition statement in which it occurs; the permitted extent of truncation is shown in the individual member definition statement specifications in "[Data Definition Statement Specifications](#)" on page 91 and in the various interface facility publications. Truncation limits indicated in the clause specifications in the pages of this section are those beyond which further truncation would cause ambiguity with other common clause keywords: these limits may indicate greater extents of permitted truncation than those applying in the context of particular member definition statements.

Common clauses provide a convenient method for differentiating between different types of documentation associated with a member data definition. CATALOGUE and ALIAS clauses may be listed with the dictionary mode LIST command.

ACCESS-AUTHORITY Clause

The ACCESS-AUTHORITY clause defines the persons, functions, or departments that have access to the real data to which the member relates, together with the nature of that access.

ACCESS-AUTHORITY Format

```

ACCESS-AUTHORITY 'name' [ { READ-ONLY
                           UPDATE
                           SECURITY CONTROL } ]
[ , 'name' [ { READ-ONLY
              UPDATE
              SECURITY-CONTROL } ] ] ...

```

where *name* is a name not more than 32 characters in length, conforming to the rules for user names and owner names stated in "[Rules Governing Variables](#)" on page 19.

ACCESS-AUTHORITY Remarks

1. Not more than one ACCESS-AUTHORITY clause can occur in any member definition statement.
2. The keyword READ-ONLY, UPDATE, or SECURITY-CONTROL, if declared, applies only to the immediately preceding name.
3. UPDATE authority is assumed to include READ authority.
4. SECURITY-CONTROL authority is assumed to include UPDATE and READ authority.

5. It is suggested that the Controller should define whether READ-ONLY or UPDATE access authority is to be assumed by users, to be the default case when none of READ-ONLY, ACCESS, and SECURITY-CONTROL is declared.

ACCESS-AUTHORITY Examples

```
ACCESS-AUTHORITY
  'ACCOUNTS' READ-ONLY,
  'PERSONNEL' UPDATE,
  'PERSONNEL MANAGER' SECURITY-CONTROL

ACCESS-AUTHORITY
  'DATABASE ADMINISTRATOR' SECURITY-CONTROL
  , 'SMITH' UPDATE, 'JONES' UPDATE, 'BROWN' UPDATE

ACCESS-AUTHORITY 'DEPARTMENTAL MANAGERS ONLY'
  , 'PERSONNEL MANAGER' SECURITY-CONTROL
  , 'A.SMITH OR J.BROWN' UPDATE
```

ADMINISTRATIVE-DATA Clause

Use the ADMINISTRATIVE-DATA clause to include in the documentation of the member any plain language information required by the user regarding the administration and control of the real data or of the data definition.

ADMINISTRATIVE-DATA Format

```
ADMINISTRATIVE-DATA 'text'
```

where *text* represents up to 32,767 delimited character strings, each string having up to a maximum of 252 characters.

Note: _____

Not more than one ADMINISTRATIVE-DATA clause can occur in any member definition statement.

ADMINISTRATIVE-DATA Examples

```
ADMINISTRATIVE-DATA 'SPECIFIED BY:  A.JONES '
                    'CODED BY      :  B.SMITH '
                    '                :  J.ARTHUR '
                    'TEST DATA    :  A.JONES '
```

```
ADMINISTRATIVE-DATA
'BATCHED INPUT FORMS WITH CONTROL TOTALS TO BE SENT FROM'
'PROGRESS DEPT TO DATA CONTROL BY 4P.M. DAILY'
```

ALIAS Clause

Use the ALIAS clause to state alternative names or identification for the member.

ALIAS Format

```
ALIAS 'alias' [, 'alias'] ...
```

where *alias* is a string of from one to 79 printable characters. A space (hexadecimal 40) is considered as a printable character.

ALIAS Remarks

1. Aliases can be used to state alternative names or identification by which a member is known in different programming languages, in different natural languages, in different systems, in clerical or end user departments, or in any other context.
2. A maximum of sixteen aliases can be declared. If two or more aliases are listed in the clause, each except the last in the list must be followed by a comma. Entries in the list may additionally be separated by spaces.
3. There are no restrictions on the character string that is the alias, other than the restriction to a maximum of 79 printable characters. Thus it is permissible for an alias to be the same as the name of a member, or a catalogue classification, or a password, or any other name recorded in the modeling dictionary; or none of these. If an alias is the same as the name of another member of the modeling dictionary, no relationship to that member is established or implied. The only relationship recorded for an alias is to the member in whose ALIAS clause the alias appears.
4. When an alias is encountered for the first time on encoding any member, a data entries record is created for that alias. If the alias is subsequently encountered on encoding another member, the alias record is updated in respect of that member; a second data entries record is not created for that member.
5. Not more than one ALIAS clause can occur in any member definition statement.
6. Aliases can be processed by the dictionary mode command LIST.

ALIAS Examples

```
ALIAS 'ARTICLE NUMBER', 'PART-NUMBER'
```

CATALOGUE Clause

Use the CATALOGUE clause to classify a member.

CATALOGUE Format

```
CATALOGUE 'classification' [, 'classification'] ...
```

where *classification* is a string of from one to 79 printable characters. A space (hexadecimal 40) is considered as a printable character.

CATALOGUE Remarks

1. Any number of classifications up to a maximum of 32,767 can be declared as applying to a member. If two or more classifications are listed in the clause, each except the last in the list must be followed by a comma. Entries in the list may additionally be separated by spaces.
2. Classifications can be processed by the dictionary mode command LIST.
3. The catalogue facility can be used to categorize members of the modeling dictionary according to system or application areas, or to end user or originating departments.
4. Not more than one CATALOGUE clause can occur in any member definition statement.
5. When a classification is encountered for the first time on encoding any member, a data entries record is created for that classification. If the classification is subsequently encountered on encoding another member, the classification record is updated in respect of that member; a second data entries record is not created for the classification.

CATALOGUE Examples

```
CATALOGUE 'PAYROLL' , "COSTING",  
          "MACHINE SHOP", 'R. AND D.'
```

COMMENT Clause

Use the COMMENT clause to include plain language comments in the documentation of the member.

COMMENT Format

```
COMMENT 'text'
```

where *text* represents up to 32,767 delimited character strings, each string having up to a maximum of 252 characters.

Note: _____

Not more than one COMMENT clause can occur in any member definition statement.

COMMENT Examples

```
COMMENT "THIS MEMBER WILL NEED TO BE UPDATED"  
       "IF A BRANCH OFFICE IS OPENED"
```

```
COMMENT 'THIS IS THE THIRD VERSION OF THIS DEFINITION. IT'  
       'TAKES INTO ACCOUNT ALL COMMENTS RECEIVED UP TO'  
       '30 JUNE 1981'
```

DESCRIPTION Clause

Use the DESCRIPTION clause to include a plain language description in the documentation of a member.

DESCRIPTION Format

DESCRIPTION 'text'

where *text* represents up to 32,767 delimited character strings, each string having up to a maximum of 252 characters.

Note: _____

Not more than one DESCRIPTION clause can occur in any member definition statement.

DESCRIPTION Examples

```
DESCRIPTION "THIS ITEM IS THE TOTAL EX WORKS VALUE"  
           "OF THE ORDER IN LOCAL CURRENCY"
```

EFFECTIVE-DATE Clause

Use the EFFECTIVE-DATE clause to record the date on which the data definition came into effect, or is to come into effect.

EFFECTIVE-DATE Format

EFFECTIVE-DATE *date*

where the format of *date* is as determined by the DCUST installation macro. If DCUST has not been tailored by the Controller, then the default format applies; that is:

day month year

where:

day is one or two numeric characters.

month is either one or two numeric characters, or one of the following undelimited character strings of three characters (which may be truncated to the extent required):

| | | | | | |
|------------|------------|------------|------------|------------|------------|
| <u>JAN</u> | <u>FEB</u> | <u>MAR</u> | <u>APR</u> | <u>MAY</u> | <u>JUN</u> |
| <u>JUL</u> | <u>AUG</u> | <u>SEP</u> | <u>OCT</u> | <u>NOV</u> | <u>DEC</u> |

year is two or four numeric characters.

The elements *day*, *month*, and *year* are separated by any non-alphanumeric printable character. A space character is regarded as a printable character. If a semicolon, comma, or right parenthesis is used as the separator character, then date must be enclosed within quotes. If the separator character is any other acceptable character, data may be optionally enclosed in quotes.

Note:

Not more than one EFFECTIVE-DATE clause can occur in any member definition statement.

EFFECTIVE-DATE Examples

The second and third of these examples are valid only if the installation macro DCUST is tailored to accept the particular date formats illustrated.

EFFECTIVE-DATE ' 30 JUN 81 '

EFFECTIVE-DATE 810630

EFFECTIVE-DATE ' 6 . 30 . 1981 '

FREQUENCY Clause

The purpose of the FREQUENCY clause is to define the frequency with which the data defined by the member is accessed and/or run and/or updated and/or backed up.

FREQUENCY Format

$$\text{FREQUENCY} \left\{ \begin{array}{l} \textit{frequency} \\ \left\{ \begin{array}{l} \text{ACCESS} \textit{frequency} \\ \text{RUN} \textit{frequency} \\ \text{UPDATE} \textit{frequency} \\ \text{BACK-UP} \textit{frequency} \end{array} \right\} \dots \end{array} \right\}$$

where *frequency* is:

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{MONTHLY} \\ \text{WEEKLY} \\ \text{DAILY} \\ \text{HOURLY} \end{array} \right\} \left[\begin{array}{l} \textit{number} \\ \text{'string'} \end{array} \right] \\ \textit{number} \\ \text{'string'} \end{array} \right\}$$

where:

number is an unsigned integer.

string is a character string of not more than 256 characters.

FREQUENCY Remarks

1. Not more than one FREQUENCY clause can occur in any member definition statement.
2. A FREQUENCY clause can contain either:
 - A single frequency specification.
 - Any number of ACCESS, RUN, UPDATE, or BACK-UP subordinate clauses. If two or more of these subordinate clauses are present, then they may be in any order.

FREQUENCY Examples

```
FREQUENCY 'LAST FRIDAY EACH MONTH'  
FREQUENCY RUN WEEKLY ACCESS 250
```

NOTE Clause

Use the NOTE clause to include any annotation required by the user in the documentation of the member.

NOTE Format

```
NOTE 'text'
```

where *text* represents up to 32,767 delimited character strings, each string having up to a maximum of 252 characters.

Note: _____

Not more than one NOTE clause can occur in any member definition statement.

NOTE Example

```
NOTE 'THIS IS ONLY AN EXAMPLE'  
  
NOTE "INPUT DATA FOR 'SAMIS' "  
"SUITE OF PROGRAMS"
```

OBSOLETE-DATE Clause

Use the OBSOLETE-DATE clause to record the date on which the data definition became or is to become obsolete.

OBSOLETE-DATE Format

```
OBSOLETE-DATE date
```

where the format of *date* is as determined by the DCUST installation macro. If DCUST has not been tailored by the Controller, then the default format applies; that is:

```
day month year
```

where:

day is one or two numeric characters.

month is either one or two numeric characters, or one of the following undelimited character strings of three characters (which may be truncated to the extent required):

JAN FEB MAR APR MAY JUN
JUL AUG SEP OCT NOV DEC

year is two or four numeric characters.

The elements *day*, *month*, and *year* are separated by any non-alphanumeric printable character. A space character is regarded as a printable character. If a semicolon, comma, or right parenthesis is used as the separator character, then date must be enclosed within quotes. If the separator character is any other acceptable character, data may be optionally enclosed in quotes.

Note: _____

Not more than one OBSOLETE-DATE clause can occur in any member definition statement.

OBSOLETE-DATE Examples

The second and third of these examples are valid only if the installation macro DCUST has been tailored to accept the particular date formats illustrated.

OBSOLETE-DATE 1/JAN/1982
OBSOLETE-DATE '1 JANVIER 82'
OBSOLETE-DATE JA-01 -82

QUERY Clause

Use the QUERY clause to enable any questions to answer, or any undecided points to resolve, to record within the data definition under development; or to record details of any information source to which reference may be made in the event of any questions about the data definition arising.

QUERY Format

QUERY ' *text* '

where *text* represents up to 32,767 delimited character strings, each string having up to a maximum of 252 characters.

Note: _____

Not more than one QUERY clause can occur in any member definition statement.

QUERY Examples

```
QUERY '5 NUMERIC-CHARACTERS OR 6? '  
  
QUERY 'HAS THE CONTAINS CLAUSE BEEN AGREED BY MARKETING '  
  
QUERY 'ARE ALPHABETIC DESCRIPTORS ACCEPTABLE '  
  
QUERY  
'REFER ANY QUESTIONS TO T.BAKER, PRODUCT DEVELOPMENT '
```

SECURITY-CLASSIFICATION Clause

Use the SECURITY-CLASSIFICATION clause to define the security classification of the real data to which the member relates, or to record the security history of the real data.

SECURITY-CLASSIFICATION Format

```
SECURITY-CLASSIFICATION 'string' [FROM date]  
[, 'string' [FROM date]]...
```

where:

string is a character string of not more than 256 characters.

date is in a format determined by the DCUST installation macro. If DCUST has not been tailored by the Controller, then the default format applies; that is:

day month year

where:

day is one or two numeric characters.

month is either one or two numeric characters, or one of these undelimited character strings of three characters (which may be truncated to the extent required):

| | | | | | |
|------------|------------|------------|------------|------------|------------|
| <u>JAN</u> | <u>FEB</u> | <u>MAR</u> | <u>APR</u> | <u>MAY</u> | <u>JUN</u> |
| <u>JUL</u> | <u>AUG</u> | <u>SEP</u> | <u>OCT</u> | <u>NOV</u> | <u>DEC</u> |

year is two or four numeric characters.

The elements *day*, *month* and *year* are separated by any non-alphanumeric printable character. A space character is regarded as a printable character. If a semicolon, comma or right parenthesis is used as the separator character, then date must be enclosed within quotes. If the separator character is any other acceptable character, data may be optionally enclosed in quotes.

SECURITY-CLASSIFICATION Remarks

1. Not more than one SECURITY-CLASSIFICATION clause can occur in any member definition statement.
2. The FROM clause specifies the date on which the security classification stated in the associated 'string' became (or is to become) effective. A later date specified in a FROM clause associated with another string may specify the date on which the earlier dated 'string' became (or is to become) superseded, or may state the effective date of a parallel classification, or other details. This depends on the security system defined for the real data in the particular user installation; the SECURITY-CLASSIFICATION clause must be interpreted in relation to that system.

SECURITY-CLASSIFICATION Examples

```
SECURITY-CLASSIFICATION 'COMPANY SECRET', 'ENCRYPTED'
```

```
SECURITY-CLASSIFICATION 'TOP SECRET' FROM 800615,  
'SECRET' FROM 810102, 'CONFIDENTIAL' FROM 810703
```

```
SECURITY-CLASSIFICATION 'DERESTRICTED' FROM JAN/1/1981,  
'RESTRICTED' FROM JAN/1/1979
```

The date formats in the second and third of the above examples are valid only if the installation macro DCUST has been tailored to accept those particular formats.

SEE Clause

Use the SEE clause to include cross references to other members in the documentation of the member.

SEE Format

```
SEE member-name [FOR 'string']  
[,member-name [FOR 'string']]...
```

where:

member-name is the name of a member of the modeling dictionary.

string is a character string of not more than 256 characters.

SEE Remarks

1. Not more than one SEE clause can occur in any member definition statement.
2. If, when the member containing the SEE clause is successfully encoded, member-name has no data entries record, DesignManager creates a dummy data entries record for member-name. The dummy record is created as a dummy of the same member type as the member that contains the SEE clause.

SEE Example

```
SEE PERSONNEL-RECORDS FOR 'ADMINISTRATIVE-DATA'
```

5

DesignManager Output

This chapter provides information on DesignManager output in general. These are the categories of output which DesignManager produces:

- Messages
- Echoed input lines
- Dictionary mode output
- Design mode output, which can be subdivided into these categories:
 - Ancillary output
 - Report output

Other categories are produced by these optional additional facilities:

- Interactive Front-end (selectable unit DSR-FE30)
- User Printer Graphics (selectable unit DSR-UD31)

These are documented in [Chapter 9, "Interactive Front-end Facility," on page 179](#) and [Chapter 10, "User Printer Graphics \(DSR-UD31\)," on page 209](#), respectively.

The remainder of this section includes descriptive remarks common to all of the above categories.

DesignManager output can be written to a line printer, a magnetic tape, a magnetic disk, or a terminal.

When operating in batch, DesignManager writes out the input commands; other input lines associated with some of the commands are also written out. When DesignManager operates interactively, and the same device is used for input and for output, the input lines already appear on the terminal's print or display, so they are not repeated as output.

When output writes to a magnetic tape or disk, each output record contains a print line image, including the print control characters. In the discussions that follow, it is assumed that output is to a line printer.

The print line width is 132 characters.

When output writes to a printer or to a terminal under any operating system or teleprocessing monitor, if the output device's line length is less than the print line width, DesignManager will, if necessary, adjust the print onto two or more lines per DesignManager output line.

Each page of DesignManager output is headed by a line containing the date, the time at which the run started, the release number of the DesignManager program used, and the page number within the run. On the second and subsequent pages, the name of the modeling dictionary currently open is printed on a second heading line. In Integrated Manager Products installations with the ASG-DataManager Status facility (selectable unit DMR-DD2) installed, the name of the current status is also printed on the second heading line.

DesignManager throws to a new page:

- After the nth print line of a page, where n is a number determined by the value of the PAGE keyword of the LOPT1 installation macro (see the appropriate Installation publication for DOS or OS environments).
- To commence a report.
- When the User Formatted Output facility (selectable unit DSR-UD30) is installed, in the same circumstances as above and also according to formatting specifications.

Messages

Each message has an identifier which is unique in the Manager Product range. This identifier is printed on the extreme left of the output line followed by the body of the message. The identifier consists of these three components:

- The prefix DM
- A five-digit number
- A suffix of I, W, E, S, or C

The prefix denotes Diagnostic Message output by a Manager Product.

The five-digit number enables the message to be located easily in the *ASG-Manager Products Message Guide*. In that publication, all DesignManager informatory warning and error messages are listed, together with an explanation of the circumstance that gave rise to the message, and where applicable, a suggestion as to remedial action to be taken.

The suffix, termed the severity code, denotes the type of output message:

| Severity Code | Output Message |
|---------------|---|
| I | denotes an informatory message; for example, a message that a command has been successfully performed. |
| W | denotes a warning message; for example, a message that the performance of a command has resulted in a condition that may not exactly reflect the user's intentions, and which the user should therefore check. |
| E | denotes an error message. This generally relates to a syntax error in a data definition statement or command statement. Syntax checking of the remainder of the statement continues, but no major processing action is performed in respect of that statement. (If the command includes a list of member names, it is treated as a series of commands each relating to one member; so that if there is an error in respect of one member name, for example, a name not yet present in the dictionary, processing continues with the next member in the list.) |
| S | denotes a serious error message. Processing of the command is abandoned. The run continues with the next command. |
| C | denotes a critical error message. The error results in the termination of the run. |

Echoed Input Lines

Each echoed input line is identified by a five-digit number prefix, which is followed by the body of the input line (or by text relating to the input line). The manner in which the number relates to an input line depends on the nature of the output line in which the number appears; thus:

- All lines input during the run, whether lines of commands or of data definition statements, are counted, commencing with 00001.
- If an input line is a command line or an amendment line or an amendment terminator line, it is printed to the right of the five-digit input line number.
- If an input line is a data definition statement line other than a terminator line, it is printed to the right of the five-digit number. The five-digit number contains the line number of the data definition line as recorded in the member's source record. Data definition terminator lines are not printed but are included in the input line count.
- As part of the action of a MODIFY command, the member's source record as it is held on completion of the command is printed to the right of the five-digit number. The five-digit number contains the line numbers of the data definition lines as recorded in the source record. These lines are not included in the run's input line count.
- On some error message lines, the line number of the input line to which the message relates is repeated, followed by a number indicating the incorrect element within the line. Thus,

00118/002

indicates that the second element on input line number 118 is in error.

Dictionary Mode Output

DesignManager dictionary mode output includes these commands:

- LIST command output
- PRINT command output

These commands are described in detail in ["Primary Command Specifications" on page 28](#). In dictionary mode, all remarks in ["Echoed Input Lines" on page 111](#) are also applicable.

When running in an integrated environment with ASG-DataManager, dictionary mode output is the same as that produced by ASG-DataManager, except that extensions to the output commands are available, as indicated in ["Effects on DataManager Commands" on page 172](#), to cater for DesignManager-specific member types and keywords.

Output produced by DesignManager when in dictionary mode is different from the output produced in design mode:

- It cannot be reformatted with the optional additional facility User Formatted Output (selectable unit DSR-UD30).
- In an integrated environment, the REPORT command produces output relating to members in the modeling dictionary but not to data in the workbench design area. This output differs from the output from the design mode REPORT command which produces reports on data in the workbench design area, see ["Design Reports" on page 113](#) for a detailed description of the output from the design mode REPORT command.

Design Mode Output

Design mode output falls into two main categories:

- Ancillary output
- Design reports

Ancillary output is any design mode output not produced by the REPORT or DESIGN commands (or the PLOT command when the optional additional facility User Printer Graphics, selectable unit DSR-UD31, is installed). This output cannot be reformatted by the optional additional facility User Formatted Output (selectable unit DSR-UD30).

Design reports form the main body of reports required for DesignManager data analysis, data modeling and logical schema evaluation and design iteration. These reports describe data in the workbench design area, both before and after normalization. When the optional facility User Formatted Output is installed, these reports can be reformatted by the user. If the User Printer Graphics facility is installed, then graphic output of the workbench design area can be produced from the workbench design area using the PLOT command. When both User Printer Graphics and User Formatted output facilities are installed, then it is also possible to reformat the graphic output produced by the PLOT command.

Design Reports

These are eight design reports which DesignManager produces:

- The Data-view Report
- The Userview Report
- The Data Element Usage Analysis Report
- The Logical Schema Report
- The Intersecting Data Element Report
- The Design Audit Report
- The Entity Report (if the Enterprise Modeling facility (selectable unit DSREM10) is installed)
- The Load Factor Analysis Report (if the Load Factor Calculation facility (selectable unit DSR-PH10) is installed)

The Entity Report is discussed below in the description of the Data-view Report and the Load Factor Analysis Report is described in *ASG-DesignManager Load Factor Calculation*.

The design reports are produced using the design mode REPORT command, with the exception of the Design Audit Report which is an option of the DESIGN command. (See ["Design Mode Command Specifications" on page 73](#).)

Graphical output which can be produced with the User Printer Graphics optional additional facility is described separately in [Chapter 10, "User Printer Graphics \(DSR-UD31\)," on page 209](#).

Besides being formattable, a major feature of these reports is their ability to track information from source input to final output results and vice versa. For example, consider the Logical Schema Report, one subsection shows, for a particular normalized relation, all the dependencies which have gone into forming the relation. Additionally, the report shows, for each dependency, which data-views contained that dependency, and the relative position of that dependency within the data-views. So, it is easy, when analyzing the final relation, to keep track of exactly which data-views contributed to its formation.

Another significant feature is the ability to select subsets for the reports. Thus, in a Data Element Usage Analysis Report, it is possible to select for reporting those data elements which are of special significance to the user.

All the reports described in this section are available in two formats, either summary or detail.

The term *report category* used in this chapter is defined in the design mode REPORT command specifications (see ["Design Mode Command Specifications" on page 73](#)).

The Data-view, Userview, and Entity Reports

These reports describe all the data in the workbench design area relevant to the selected data-views (data-view is the generic term for userviews and entities). As they are identical in structure, a common description of all three reports is given.

The Summary Reports lists the total number of this data:

- Data-views (either userviews, entities, or both according to the report category, see ["REPORT" on page 84](#))
- Userviews
- Entities
- Dependencies
- Functional dependencies
- Multivalued dependencies
- Data elements
- Data elements appearing on the left-hand side of dependencies
- Data elements not appearing on the left-hand side of dependencies

For each data-view selected, the Summary Reports shows this data:

- The type of data-view (USERVIEW or ENTITY) and name
- The relative frequency (for a userview only)
- The response time (for a userview only)
- The number of dependencies
- The number of functional dependencies
- The number of multivalued dependencies
- The number of data elements
- The number of data elements appearing on the left-hand side of the dependencies
- The number of data elements not appearing on the left-hand side of dependencies
- If the workbench design area contains normalized data:
 - The number of functional dependencies found to be redundant by the DESIGN command (and therefore having an equivalent indirect access path).
 - The number of multivalued dependencies found to be inconsistent by the DESIGN command.
 - The number of data elements found to be extraneous by the DESIGN command.
 - The number and name (if named with the NAME command) of each relation in which the data-view dependencies are directly or indirectly represented.
- The number of data-views in the workbench design area which have dependencies in common with the data-view being reported, and, for each of these *associated* data-views, its name and type (USERVIEW or ENTITY), and the number of dependencies it has in common with the data-view reported.

The Detail Report shows all of the Summary Report information, and, in addition, shows all the dependencies in the data-view. For each dependency this information is shown:

- The left-hand side and right-hand side data elements
- The type of dependency (functional or multivalued)
- The multiplicity of a multivalued dependency (userviews only)

If the workbench design area contains normalized data, each dependency may have an indication showing:

- If a functional dependency is redundant
- The extraneous data elements on the left-hand side of functional dependencies
- If a multivalued dependency is inconsistent with an existing indirect functional dependency (that is, which is found to be derivable from functional dependencies by the DESIGN command)

If the report category selected is USERVIEWS or ENTITIES, then only the appropriate type of data-view is reported on. However, an *associated* data-view (that is, a data-view having dependencies in common with the data-view being reported) may be either a userview or an entity as appropriate, regardless of the report category.

The Data Element Usage Analysis Report

The Data Element Usage Analysis Report gives details of the usage of all or a selected subset of the data elements held in the workbench design area.

The Summary Report shows the total number of:

- Data elements held in the workbench design area
- Data elements being reported

For each of the data elements selected, the Summary Report shows:

- The unique data element number which was assigned to that data element when it was added to the workbench design area.
- The data element name.
- The number of data-views that use the data element.
- The total number of dependencies containing the data element.
- The number of functional dependencies containing data element.
- The number of multivalued dependencies containing the data element.
- The number of times that the data element appears on the left-hand side of a dependency.
- The number of times that the data element appears on the right-hand side of a dependency.
- The total number of relations that contain the data element.
- The number of relations in which the data element appears as a prime data element.
- The number of relations in which the data element appears as a non-prime data element.

The Detail Report contains all the information given in the Summary Report, and additionally contains detailed information on the usage of the data element by individual data-view, by individual dependency, and by individual relation.

For each data-view using a particular data element this information is given:

- The unique data-view number which was assigned to that data-view when it was added to the workbench design area.
- For each dependency which contains the data element and is held in the data-view:
 - The relative dependency number
 - The data elements on the left-hand side of the dependency
 - The dependency type (either functional or multivalued)
 - The data elements on the right-hand side of the dependency
 - The absolute dependency number

For each dependency using a particular data element this information is given:

- The absolute dependency number
- The data elements on the left-hand side of the dependency
- The dependency type (either functional or multivalued)
- The data elements on the right-hand side of the dependency
- A list of the data-views containing the dependency. For each of the data-views, the list gives this information:
 - The relative dependency number in the data-view
 - The data-view name

For each relation using a particular data element this information is given:

- The relation number
- The relation name (if named using the NAME command)
- The role of the data element in the relation (that is, either whole key, part key, or non-key)

The Logical Schema Report

This report describes the logical schema generated by DesignManager during execution of the DESIGN command. The logical schema is the collection of (third normal form) relations present in the workbench design area following execution of the DESIGN command.

Individual relations may be specified for selective reporting, either by name (if a name has been assigned via the NAME command) or by workbench number.

Ordering of the report is also possible by means of the ALPHA keyword (see the design mode REPORT command specifications in ["Design Mode Command Specifications" on page 73](#)).

After displaying the total number of relations in the workbench design area, the Summary Report shows for each relation selected:

- The number of the relation
- The name of the relation (if named using the NAME command)
- The data elements forming the primary key of the relation
- The non-key data elements of the relation (if any are present)

The Detail Report shows, for each relation selected, all of the above information. In addition, for each dependency represented by the relation, the Detail Report shows:

- The absolute dependency number.
- The left-hand side and right-hand side data elements.
- The type of dependency (functional or multivalued).
- The origin of the dependency, as either implied or having been obtained from one or more data-views, in which case the name and type (USERVIEW or ENTITY) of the data-view and the relative number of the dependency in each is also output.

The Detail Report shows the identification of the directly associated target relations pointed to by the reported relation. Two types of pointers are possible, *one* and *many* (representing foreign key and hierarchical key associations, respectively), where R1 is the relation being reported:

- A *one* pointer is defined from R1 to R2 if the key of R2 is a foreign key in R1 (possibly appearing as a subset of R1's key) and there is no other relation R3 such that both:
 - The key of R3 is a foreign key in R1
 - The key of R2 is contained in the key of R3

Consider, for example, the relations:

R1 (A,B,C) with key A

R3 (B,C,D) with key B+C

R2 (B,E) with key B

Note that relation R1 has foreign keys both to R2 and to R3. Nevertheless, a *one* pointer from R1 would be generated only to R3, because R3 in turn has a one-pointer to R2; thus a further one-pointer from R1 to R2 would be redundant. (R2 has a *many* pointer to R3; see below for definition.)

- R2 is a target relation of R1 associated by a *many* pointer if the key of R1 is contained in the key of R2, and there is no other relation R3 such that:
 - The key of R1 is contained in the key of R3.
 - The key of R3 is contained in the key of R2.

As in the case of *one* pointers, no skipping over levels is depicted.

In the Logical Schema report, a *many* pointer in one direction implies a *one* pointer in the reverse direction. The converse is not necessarily true. In the example above, relation R2 has a *many* pointer to R3 and thus R3 has a *one* pointer to R2. However, although R1 has a *one* pointer to R3, there is no *many* pointer in the reverse direction. For each directly associated target relation, this information is given:

- The type of pointer (*one* or *many*)
- The number of the associated relation
- The name of the associated relation (if one has been assigned using the NAME command)
- The data elements forming the primary key of the associated relation

The Detail Report shows the data elements of the relation arranged, with all appropriate keywords, according to the USERVIEW member definition syntax for one of these:

- A functional dependency if the relation is an FD relation.
- A multivalued dependency if the relation is an MVD relation.
- Two or more multivalued dependencies if the relation is derived from the merging of MVD relations with identical keys (see the DESIGN command specifications and ["Generating a Third Normal Form Relational Schema" on page 132](#)). A separate multivalued dependency is created for each merged relation.
- A combination of the above if the relation derives from the merging of an FD relation and one or more MVD relations all with identical primary keys.

These are the dependencies represented by the relation being reported and will already have been output in a previous sub-section, as described above.

The syntax generated for each USERVIEW definition is prefixed by the command identifier *ADD*, followed by the relation name (if the relation has been named with a NAME command) and a blank space. This part of the output is intended for use in storing the relations in the modeling dictionary as USERVIEW members (see [Chapter 6, "DesignManager Procedures and Use," on page 123](#) and, in particular, ["Naming Relations and Storing Them in the Dictionary" on page 144](#)).

The Intersecting Data Elements Report

This report, which must be produced in its entirety, displays all the intersecting data elements (if any) in the workbench design area. An intersecting data element is one which appears as the right-hand side of more than one functional dependency and is not contained in the left-hand side of any. This means, in effect, that the data element is defined uniquely by more than one left-hand side. The condition that it, must not itself be a left-hand side, expresses the fact that it cannot be the key of a relation. This is because, if it were, then its appearance on the right-hand side of more than one functional dependency, would merely be representing the fact that it appears as a foreign key in more than one relation. This latter condition is quite common in a logical schema and so is not reported. The occurrence of a data element in the right-hand side of more than one functional dependency when it is not itself a left-hand side, however, indicates the potential presence of a homonym or synonym situation.

These two examples illustrate this:

Example 1. Consider this intersecting data element STUDENT-ADDRESS:

```
STUD-NAME -----> STUDENT-ADDRESS
STUDENT-NAME -----> STUDENT-ADDRESS
```

It is clear that here STUD-NAME and STUDENT-NAME are synonyms—different names used to mean the same thing. Corrective action must be taken to make both left-hand sides the same.

Example 2. Consider this intersecting data element ADDRESS:

```
CUSTOMER-NUMBER -----> ADDRESS
SHIPMENT-NUMBER -----> ADDRESS
```

What has happened here, is that the two addresses should be different. The first is a corporate headquarters address, while the second is really a shipment or warehouse address. This is a case of a homonym—the same name (ADDRESS) used to mean different things. Corrective action to be taken here is clearly to differentiate between the two addresses by giving them different names.

Not all intersecting data elements represent anomalies. Some may be quite valid as shown here:

```
EMPLOYEE-NUMBER -----> EMPLOYEE-NAME
SOCIAL-SECURITY-NUMBER -----> EMPLOYEE-NAME
```

Both are correct and will give rise to two relations which can be merged with a selection of one key being made for the resulting relation.

When the right-hand side is itself a left-hand side, no anomaly exists:

```
PROJECT-NUMBER -----> DEPARTMENT
EMPLOYEE-NUMBER -----> DEPARTMENT
DEPARTMENT -----> FUNCTION
```

This situation gives rise to three relations and merely expresses the fact that projects and employees are related to one department.

An intersecting right-hand side never consists of more than one data element. Functional dependencies with more than one data element on the right-hand side (as entered in userviews), are broken down into *elemental functional dependencies* with single data elements on the right-hand side.

The Intersecting Data Elements Report can be obtained either before or after normalization. In general, there are differences in the report before and after normalization due to the removal of redundant functional dependencies. Here is an illustration:

```
A -----> C
B -----> C
A -----> X
X -----> C
```

Before normalization, C is an intersecting data element with A, B, and X on the left-hand side. After normalization, the functional dependency:

A -----> C

is removed as redundant because of:

A -----> X

and

X -----> C

C now only appears on the right-hand side of the two functional dependencies:

B -----> C

and

X -----> C

A further discussion on intersecting data elements is given in the latter part of "[Merging Userviews into the Workbench Design Area](#)" on page 130.

The Summary Report shows, the usage of each intersecting data element in the workbench design area, and for each functional dependency in which it appears as the right-hand side.

- The left-hand side data elements of the dependency, and the number of the dependency in the workbench design area (the absolute dependency number).

The Detail Report shows, for each intersecting data element, all of the above information, and, for each dependency reported, either:

- The fact that the dependency is implied.
- The name and type (USERVIEW or ENTITY) of each data-view from which the dependency was obtained and the relative number of the dependency in that data-view.

The Design Audit Report

This report, unlike the ones described in preceding sections, is produced not by the design mode REPORT command, but by the design mode DESIGN command with the AUDIT option. The report is produced during the execution of the DESIGN command to give a complete audit trail of the effects of normalization (see "[The DesignManager Design Procedure](#)" on page 138). It allows the user to analyze the effects of the DESIGN command in depth. Any undesirable effect can be easily traced back to a specific dependency and hence to its originating data-views, either userviews or entities.

Both Summary and the Detail reports consist of these three parts:

- A list of functional dependencies with extraneous data elements in the lefthand side. The extraneous data elements are indicated by asterisks.
- A list of functional dependencies removed because they are redundant.
- A list of multivalued dependencies found to be inconsistent with derivable functional dependencies.

For each dependency reported, the output includes the absolute dependency number and the origin of the dependency. This origin may have been *implied* (see the design mode "[MERGE](#)" on [page 79](#)) or have been obtained from one or more data-views. In the latter case the output also includes:

- Identification of each data-view by name and by type as either a USERVIEW or an ENTITY.
- The relative number of the dependency in that data-view.

The above information constitutes the Summary Report. The Detail Report shows, in all three parts, the above information and, for each data element or dependency removed, this information:

- An *audit trail* of functional dependencies which led to the elimination of the relevant data.
- For each dependency in this audit trail, its absolute number and details of its origin (as above).

Any information removed from the workbench design area is only removed logically. The data is retained physically and prior to execution of the next MERGE command, the workbench design area is first restored to its *unnormalized* state. This is to ensure that any subsequent normalization is performed on the entire data in the workbench design area.

Each of the three parts of the report appear only if relevant data has been removed during normalization. If no data has been removed, no report is produced.

6

DesignManager Procedures and Use

The steps involved in using DesignManager to generate a logical database design model as a third normal form (3NF) relational schema were described in [Chapter 1, "Introduction to DesignManager," on page 1](#) and are summarized here:

- Definition and storage in the modeling dictionary of data elements required for the database.
- Definition and storage in the dictionary of userviews, in the form of dependencies which are to be represented in the design, and are expressed as:
 - A user's requirements for accessing data elements
 - A user's perception of the logical associations holding amongst them
- Merging of userviews to form a single *composite view* by cumulative loading from the dictionary into the workbench design area.
- Normalization of the composite view and generation of a 3NF relational schema in the workbench design area.

DesignManager reports on the content of the dictionary and workbench design area are available at all stages of the design process. In these sections, a more detailed discussion is given of the above procedures and some techniques which can be helpful when using DesignManager to generate a logical database design.

Formulation of Userviews

This section describes the structure and definitions of userviews, and their expression of functional and multivalued types of dependencies. It also provides instruction of how to develop, name, and adopt a top-down approach to the formulation of userviews.

Structure and Definitions

A userview expresses a user's requirements for accessing the values of the database data elements in a particular application. It is formulated in terms of data dependencies which effectively represent required access paths between individual data elements or sets of data elements. A userview is regarded as a list of dependencies of the form *data element (or set of data elements) determines data element*. Dependencies are also used to express logical associations between data elements or sets of data elements which can be included in userviews along with the user's access requirements. In addition to a list of dependencies, a userview definition may also contain the user's estimate of the frequency with which the data elements will be accessed and the response time requirements. After the merging of userviews from the modeling dictionary into the workbench design area, these values are used to calculate the total access frequency and the weighted average response time for each data element over all the userviews (in the workbench design area) in which it appears.

Two types of dependencies can be expressed in a userview, functional (or single-valued) dependencies, and multivalued dependencies. A dependency is *functional* from A to B and written as shown here, if each value taken on by A (or set of values if A is a set of data elements) determines exactly one value of B:

A ---> B

If each value of A determines a variable number of values of B (0, 1, or more), the dependency is *multivalued* and is written:

A --->> B

To each EMPLOYEE of a company, for example, there corresponds the employee's unique home ADDRESS, represented by the functional dependency (FD):

EMPLOYEE ---> ADDRESS

Since each EMPLOYEE, however, may have more than one CHILD (or none), the correspondence between EMPLOYEE and CHILD would be represented by the multivalued dependency (MVD):

EMPLOYEE --->> CHILD

Note: _____

Although an FD is in fact a special case of an MVD, they are treated separately by the DesignManager software and affect the design in different ways. The distinction between the two at the input stage is important.

In DesignManager, data elements are defined and stored in the modeling dictionary as ITEM and GROUP members. The dependencies relating the data elements which are relevant to a particular user are defined and held in the dictionary within USERVIEW members.

One level higher in the dictionary hierarchy, the VIEWSET member represents a collection of USERVIEWs (USERVIEW and/or VIEWSET names may be specified in the CONTAINS clause). The VIEWSET member type is provided as a convenience to the user to avoid the tedious job of listing a large number of USERVIEWs in the MERGE command when merging userviews from the modeling dictionary into the workbench design area: only a single VIEWSET need be named.

Userview Development

Userviews may be developed in a variety of ways, usually by the data administrator in collaboration with the users affected. The requirements can be obtained from reports, forms, files, screens, interviews, or by a combination of these. After the application or situation necessitating user access to the database is identified, a userview is usually developed using these two major steps:

1. Identify the set of data elements required and name them (for storage in the modeling dictionary as ITEMS or GROUPs) in a consistent fashion.
2. Determine the dependencies that relate them in the user's application.

These steps are discussed in some detail in ["Naming the Data Elements" on page 125](#) and ["Determining the Dependencies" on page 128](#).

Naming the Data Elements

Use either of these commands to determine whether or not a name selected for a data element has already been used for a previously entered dictionary member (see ["Primary Command Specifications" on page 28](#)):

```
LIST MEMBERS 'member-name' ;  
PRINT 'member-name' ;
```

A user may find that the data element required has already been entered or, on the other hand, that a different name must be chosen to avoid the occurrence of a homonym; that is, where a data element name is used in more than one user's application, each using that data element to represent a different set of values. For example, the DESCRIPTION clause (see ["Common Clause Specifications" on page 99](#)) of a member named EMPL-SALARY may specify that this data element represents an employee's net annual salary: to represent gross annual salary, a different data element and name should be chosen (see further discussion of the homonym problem in ["Use of Pseudo-FDs to Handle Subcategories" on page 151](#)).

The more difficult task, especially in the case of a large database, is to avoid the occurrence of synonyms when naming data elements and storing them in the dictionary. If, for example, EMPLOYEE-NUMBER and EMPL-NO were entered in the dictionary as different members (presumably by different users) representing the same data element, the software would treat them as different data elements and could produce results which would be misleading.

Before naming and storing a data element in the dictionary, a user can check for possible synonyms by entering the dictionary mode command shown here, which lists every member (by name and member type) whose name contains the specified string:

```
LIST WHEN ANY = 'string' ;
```

Consider the case of the user who selects the name EMPLOYEE-NUMBER for a data element. The user is able to detect any dictionary member names containing the string EMP, such as EMPLOYEE-NO, EMPL-NUMBER, or EMP-NAME (or EMPLOYEE-NUMBER itself) by entering this command:

```
LIST WHEN ANY = 'EMP' ;
```

More detailed information on a particular name LISTed is obtained by further entering this command:

```
PRINT 'member-name' ;
```

By following the above procedure, a user usually can see if the same data element has been entered previously to the modeling dictionary under a different name. With regard to avoiding possible homonyms and synonyms, it is recommended that every data element entered in the dictionary include a DESCRIPTION clause, specifying the particular data values that it represents, plus a NOTE clause, containing both the name of the user who entered it and the date it was entered (see ["Common Clause Specifications" on page 99](#) for DESCRIPTION and NOTE clause specifications).

A data element normally is defined as an ITEM member of the modeling dictionary. Only use a GROUP member when the data element being defined has two or more components, some of which are themselves to be used as data elements in userviews. For example, DATE should be defined as a GROUP member CONTAINing DAY, MONTH, and YEAR if both DATE and any of the ITEMS DAY, MONTH, and YEAR are specified as data elements in a userview.

To facilitate the above procedure (and in some instances make it unnecessary), it is recommended that company-wide naming standards and conventions be adopted. In addition, when defining and storing members in the modeling dictionary, include associated ALIAS and CATALOGUE clauses (see ["Common Clause Specifications" on page 99](#)) which reflect these standards. The following approach illustrates some of the techniques available. Basic to the procedure is maintenance of a company list of standard keywords with required abbreviations for their use in data element names. The list can be conveniently stored in the NOTE clause of an appropriately named (for example, ABBREVIATION-LIST) ITEM member of the modeling dictionary and retrieved via the PRINT command. If EMPLOYEE and NUMBER, for example, were keywords in the list with respective abbreviations EMP and NO, then the LIST WHEN ANY = 'string' procedure described above probably could be skipped since reference to the list would indicate that the only name that should have been selected by another user is EMP-NO. Whether or not it has been defined previously can be determined by entering a PRINT EMP-NO command to the

dictionary. If not, the user would enter EMP-NO as an ITEM in the dictionary and, for purposes of identification and classification, the user could include an ALIAS clause containing the expanded form EMPLOYEE-NUMBER of EMP-NO and the CATALOGUE classifications EMPLOYEE and NUMBER. Even if EMPLOYEE and/or NUMBER (or recognizable synonyms) were not keywords in the abbreviation list, it would not be necessary for the user to enter a LIST WHEN ANY = 'string' command (assuming rigid enforcement of company naming standards). By inspection of the list, the user would know that no employee identification data element had been entered previously to the dictionary. The user could then obtain approval for the new keywords EMPLOYEE and NUMBER and their abbreviations, add them to the abbreviation list, and store the new data element EMP-NO as a new ITEM member.

Although the LIST command would not have been required in the relatively simple case illustrated above, the command can be useful in conjunction with an abbreviation list when the naming options are less clear-cut. Consider, for example, the case of naming a data element for the annual salary of an employee. A user might find the keywords ANNUAL, ANNUM, YEAR, SALARY, and EMPLOYEE, all appearing in the list, abbreviated respectively as ANN, ANN, YR, SAL, and EMP. (It would not be intrinsically wrong to assign the same abbreviation to more than one keyword provided that the meaning would always be clear in the context of a data element name. In general, however, this practice should be avoided.) By entering one or more LIST WHEN commands with compound conditions, the user could determine whether or not a data element has been defined for an employee's annual salary and stored in the dictionary, and, if it has, which keyword abbreviations have been used in the name and in what order.

This command might indicate that there is no member of the dictionary whose name contains all three specified keyword abbreviations:

```
LIST WHEN ANY = 'EMP '  
        AND ANY = 'ANN '  
        AND ANY = 'SAL ' ;
```

The user would probably conclude that, if the desired data element has already been defined in the dictionary, the abbreviation EMP was not used as part of the member name if a similar result were obtained by entering:

```
LIST WHEN ANY = 'EMP '  
        AND ANY = 'YR '  
        AND ANY = 'SAL ' ;
```

On the other hand, if this command results in a list containing the member name SAL-PER-ANN and perhaps others such as DATE-SAL-PER-ANN-EFFECT, the user could be reasonably sure that an annual salary data element is defined and stored in the dictionary as SAL-PER-ANN (and not, therefore, as ANN-SAL):

```
LIST WHEN ANY = 'ANN '  
        AND ANY = 'SAL ' ;
```

Enter this command to determine details of the member:

```
PRINT SAL-PER-ANN ;
```

This output may result from entering the preceding command:

```
PRINT OF SAL-PER-ANN
ITEM
ALIAS 'SALARY-PER-ANNUM'
CATAL 'SALARY', 'PER', 'ANNUM'
DESCR 'EMPLOYEE SALARY PER ANNUM'
NOTE 'ENTERED 17/04/82'
      'AUTHOR REH'

;
```

If, after inspection of the PRINT output, the user is still not completely satisfied that this member represents the data values required, the user can contact the author for final clarification.

All of the above procedures, with or without an abbreviation list, can be extended significantly for users in integrated DesignManager/DataManager installations. In particular, ALIASes and CATALOGUE classifications can be interrogated to determine associated member names, using the commands WHOSE ALIAS IS and WHAT FORMS respectively (see *ASG-Manager Products Dictionary/Repository User's Guide*).

Determining the Dependencies

Once the data elements are named, the next step in the development of a userview is to identify the FDs and MVDs that relate them in the user's application. For this purpose, the procedure described here is recommended. Make a list of all the data elements defined for the specific userview and, taking them one at a time, determine, for each data element in the list, which (if any) of the other data elements in the list it depends on and how, functionally or multiply. That is, taking each data element as the right-hand side (RHS) of a proposed dependency, determine which of the remaining data elements would be required on the left-hand side (LHS) to form the dependency, either as an FD or as an MVD. Here is a useful rule of thumb for this procedure:

- First attempt to form an FD, selecting as many data elements for the LHS as necessary.
- If (and only if) no FD can be formed, then see if the data element on the RHS is multiply dependent on one or more of the other data elements.

Suppose, for example, a user decides that the data elements, EMPL-NO, EMPL-NAME, SALARY, and SALARY-DATE are required for the userview SALARY-HISTORY. Using the above procedure, the user might first decide that EMPL-NO is required for primary access to some (all in the present example) of the other data elements in the userview and does not depend on any of them. Taking the other data elements in turn, the user then perhaps would identify all of these dependencies:

```
EMPL-NO                ---> EMPL-NAME
EMPL-NO + SALARY       ---> SALARY-DATE
EMPL-NO + SALARY-DATE ---> SALARY
```

These points should be noted in assessing the above selection of dependencies:

- The userview, as specified, is possibly over-determined in that only one of the last two dependencies may be required.
- The rule of thumb given above was followed in forming the last two dependencies. In each case, a sufficient number of data elements were selected for the LHS to form an FD. Thus, MVDs such as this were not included:

```
EMPL-NO ---> SALARY-DATE
EMPL-NO ---> SALARY
```

(If they were included, however, or if they were specified in a different userview by another user, the *all-key* relations generated from them via the DESIGN command would automatically be merged by the DesignManager software with the FD-relations keyed by EMPL-NO + SALARY-DATE and EMPL-NO + SALARY, respectively.)

- In general, it is advisable to select the minimum number of data elements needed on the LHS to form an FD. In many instances, however, the software is able to eliminate unnecessary data elements as extraneous automatically. For example, if, instead of the above userview, the user had entered this selection of FDs:

```
EMPL-NO ---> EMPL-NAME
EMPL-NO + EMPL-NAME + SALARY-DATE ---> SALARY
```

the data element EMPL-NAME would be removed from the LHS during the design procedure as extraneous.

Once all the dependencies required for a userview are determined, the final step is formal definition and storage in the modeling dictionary of a USERVIEW member containing the dependencies along with estimates of relative frequency and response time for the USERVIEW and of multiplicity for each MVD defined. Syntax requirements for USERVIEW members are given in ["Data Definition Statement Specifications" on page 91](#).

Note: _____

For the procedure described, not all the data elements specified in a userview need be members of the modeling dictionary. (However, each name used should previously have been analyzed for compliance with naming standards and possible synonym occurrence, as discussed in ["Naming the Data Elements" on page 125](#).) When the userview is entered in the dictionary, DesignManager creates a dummy member for each data element not already defined and reports its creation. As indicated in some detail in ["DesignManager Safety and Control Features" on page 12](#), this could indicate a synonym situation not previously detected (perhaps due to a spelling error) and which needs investigation. When it is determined that the dummy member's name is acceptable, a definition should be entered for it in the modeling dictionary as soon as possible so that other users can refer to the name.

Top-down Approach to Userview Formulation

When a considerable amount of data needs to be collected in the design of a large database, it is often useful to adopt a top-down approach to the formulation of userviews. In the context of the steps described in "[Userview Development](#)" on page 125, it is performed in conjunction with [step 1 on page 125](#), identification of data elements required for a particular application, and prior to [step 2 on page 125](#), the determination of dependencies relating the data elements. The modeling dictionary member types GROUP and ITEM are ideal aids to a top-down approach, which may be implemented through any of the accepted methodologies (BSP, ISP, Yourdon, Gane and Sarson, and Warnier-Orr, for example).

The fundamental member type for the top-down approach is the GROUP. Since a GROUP can contain both ITEMS and other GROUPs, it can be used to *collect* data elements and thus represent a rough USERVIEW before a user is ready to input a full USERVIEW definition. For example, a GROUP may be used to represent a user's form or dataflow and would contain, as ITEMS (and possibly GROUPs), the various data elements appearing in the form or dataflow. Following possible modification to the GROUP CONTAINS list, the user's requirements of these ITEMS would be expressed in the form of dependencies for input as a USERVIEW. In addition, a user may specify ITEMS (representing data elements) in a GROUP before they are defined as members of the modeling dictionary. In this case, DesignManager creates dummy ITEMS in the modeling dictionary.

Thus, a user may start with a vague idea of the data elements (and some of the dependencies relating them) required in a userview and define them as ITEMS in a GROUP. Dummy members are set up for those not existing in the modeling dictionary. Subsequently, the user may alter the members CONTAINED in the GROUP and/or put full definitions into the dictionary. When the GROUP has been refined to fit the user's requirements, dependencies can be identified relating the data elements and a USERVIEW member defined and stored in the modeling dictionary.

Merging Userviews into the Workbench Design Area

Use the MERGE command to specify one or more VIEWSET and/or USERVIEW members of the modeling dictionary to merge into the workbench design area from the dictionary. Merging is performed in cumulative fashion, the FDs and MVDs of each (directly or indirectly) specified userview are added one at a time to form a composite view of all the userviews in the workbench design area. The composite view is processed by using the DESIGN command to generate a 3NF relational schema.

The user may exercise a great deal of flexibility in the use of the MERGE command. The user may form a composite view for a design either with a single MERGE command or incrementally by using a sequence of MERGE commands (indeed, the schema itself may be generated incrementally by a series of mixed MERGE and DESIGN commands). When adopting the incremental approach, a measure of control may be introduced by the appropriate use of REPORT, LIST, and SNAPSHOT commands interspersed throughout the sequence of MERGE commands. In this way, the user can *watch* the composite view as it forms, analyze intermediate stages of its growth, and more readily spot and correct errors in the merged userviews. In particular, use of the REPORT USERVIEWS command at each stage can help significantly in forming a composite view that is complete, accurate, and free of naming anomalies.

During the merging procedure, the DesignManager software makes these modifications to the dependencies, as required:

- If a data element appears on both the left-hand side and the right-hand side of a processing dependency (FD or MVD), it is automatically removed from the right-hand side as extraneous before the dependency is added to the workbench design area. Consider this MVD:

$$A + B \text{ --->> } B + C$$

For example, the preceding MVD would change to this equivalent MVD before being added:

$$A + B \text{ --->> } C$$

If, during this process, every data element is removed from the right-hand side of a dependency (all the data elements on the right-hand side also appear on the left-hand side), then the dependency holds trivially (it is a trivial FD even if entered as an MVD) and is not added to the workbench design area. All such dependencies are automatically available to the software during the design procedure and need not be specified by a user. For example, the trivial FD would not be added:

$$A + B \text{ ---> } B$$

- After removal of extraneous data elements from the right-hand sides of dependencies, each FD whose right-hand side is still composite decomposes into its equivalent set of *elemental* FDs (see ["Glossary" on page 245](#)) before being added to the workbench design area. Consider this FD:

$$A + B \text{ ---> } C + D$$

If the preceding FD were specified in a USERVIEW being merged, these two elemental FDs would be added to the workbench design area, except as indicated in the instances described below:

$$A + B \text{ ---> } C$$

$$A + B \text{ ---> } D$$

- Duplicate dependencies are not added to the workbench design area; that is, a dependency is added the first time it is encountered, but if it appears again in a userview being merged, the duplicate is not added.
- Inconsistencies in the userviews are detected and reported. If an FD (or MVD) being processed has the same left-hand side and right-hand side as an MVD (or FD) which is present in the workbench design area, an *inconsistency* flag is set, a warning message outputs, and the FD (or MVD) is not added (see [remark 6 on page 81](#) of the MERGE command specifications for details).
- When all userviews specified in a MERGE command merge, the command generates and adds any FDs which are *implied* by the current content of the workbench design area (see [remark 7 on page 81](#) of the MERGE command).

After the MERGE command (or a sequence of MERGE commands) successfully processes, it can be useful in detecting and resolving naming anomalies to obtain a listing of all *intersecting* data elements which may be present in the workbench design area. Obtain the listing by entering this design mode command:

```
REPORT INTERSECTING-DATA-ELEMENTS ;
```

(The command is also accepted and is equally useful after a DESIGN command, although the output may differ due to the effects of normalization.) An *intersecting* data element is one which appears (in the workbench design area) as the right-hand side of more than one FD but is not contained on the left-hand side of any FD in the workbench design area. (In relational terms, it appears in more than one FD-relation and is not prime in any FD-relation; see "[Glossary](#)" on [page 245](#).)

A report on intersecting data elements can aid in the detection of synonyms and homonyms and is generally helpful in sorting out naming problems. If, for example, both of these FDs are entered, it is possible that `VENDOR` and `SUPPLIER` are synonyms:

```
VENDOR ---> PRODUCT-TYPE  
SUPPLIER ---> PRODUCT-TYPE
```

If, on the other hand, these FDs both appear, it is probable that different data element names should have been used to distinguish between managers' telephone numbers and employees' numbers:

```
MANAGER ---> TELEPHONE-NO  
EMPLOYEE ---> TELEPHONE-NO
```

Note: _____

It should be emphasized that this report is not a substitute for the procedure given in "[Userview Development](#)" on [page 125](#) for avoiding synonyms and homonyms through use of the `LIST` and `PRINT` commands and adoption of company naming standards when naming data elements. The intersecting data elements report is primarily useful in catching the odd synonym or homonym which is not detected by the earlier procedure.

Generating a Third Normal Form Relational Schema

When a set of userviews merges into the workbench design area via the MERGE command (or a sequence of MERGE commands) and the database designer is satisfied, through use of one or more REPORT commands, that the resulting composite view is well formulated, the user can enter a DESIGN command to initiate the DesignManager design procedure. The procedure is performed in these two major steps:

1. Normalization of dependencies. This step consists of simplifying the set of dependencies held the WBDA (workbench design area) by removing redundant dependencies and data elements and regrouping the remaining dependencies for automatic generation of third normal form (3NF) relations.
2. Generation of third normal form relations, one from each group of FDs and one from each MVD.

One further step taken, if necessary, is merging of relations with identical *keys*.

The set of relations generated, termed a 3NF relational schema, represents all the FDs and MVDs which were in the workbench design area when the design procedure was initiated, whether or not they were altered during execution of the DESIGN command. These are the advantages such a model provides:

- Relative stability as user requirements evolve, simplifying future changes and reducing maintenance problems in the logical and physical database structure.
- A design which, when implemented with specific values of the elements, results in a net reduction of data redundancy, consequent reduction in physical storage requirements, and elimination of most of the problems and cost associated with adding, deleting, and updating data values.

As indicated in ["Merging Userviews into the Workbench Design Area" on page 130](#), in discussing the flexible use of the MERGE command, the user also has considerable latitude in the use of the DESIGN command. It can be entered once, after all the userviews merge into the workbench design area, or can be used as part of an incremental procedure. The incremental procedure is carried out by executing a series of MERGE and DESIGN commands, interspersed, so that the design can be built up gradually. As in the case of a composite view, the design can be analyzed in each stage of its growth by appropriate use of REPORT, LIST, and SNAPSHOT commands. In addition, the PLOT command (available with the optional additional facility User Printer Graphics) can be a powerful tool in analyzing the design and re-evaluating the userviews, particularly in the final stages of growth. Use of both the REPORT USERVIEWS and the REPORT LOGICAL-SCHEMA commands provides two viewpoints for assessing the generated relations and their correspondence to the userviews merged into the workbench design area. The PLOT command outputs each generated relation and all of its pointers to other relations.

The output of the PLOT command can be used to derive a view of the *entities* inherent to the corporate enterprise being modeled and the *associations* linking them. The enterprise view, in turn can play an important role, in conjunction with high-level management consultation, in:

- Re-assessing the corporate function being modeled.
- Re-evaluating the userviews defined for this purpose.
- Modifying the userviews, if necessary, to conform to corporate standards and procedures.
- Subsequently iterating the design procedure.

A description of the top-down approach to entity modeling and the capabilities provided is given in *ASG-DesignManager Enterprise Modeling* (available only if the optional facility Enterprising Modeling, selectable unit DSR-EM10, is installed).

The output of the PLOT command can further be used to map a depicted relational schema into a first cut physical design specific to a particular database management system.

The following sections include a discussion of relations and keys, an example illustrating the benefits provided by normalization, and a description of the DesignManager design procedure. Formal definition of terms can be found in the ["Glossary" on page 245](#) and in the [Appendix, "Technical Background," on page 221](#).

Relations and Keys

The logical database design model generated by DesignManager is a 3NF schema consisting of a set of relations defined over a set of data elements. Each relation is structured for use in a physical database to depict a two-dimensional table of data values. Each column of such a table is, in concept, labeled with the name of a distinct data element and consists of values from the domain of that data element. Each row consists of a set of values, one per data element, where no two rows are identical. The rows will usually vary in number and content as entries are inserted, deleted, and updated. The structure of a relation R, on the other hand, is more or less static. It consists of a list of the data elements A1, A2, ..., An (written R (A1 , A2 , . . . , An)) and represents a set of one or more FDs and/or MVDs holding amongst the data elements.

This table, for example, is an instance of an unnormalized relation:

R (EMPLOYEE , ADDRESS , AGE , CHILD)

which could be generated from (and thus would represent) these dependencies:

EMPLOYEE ---> ADDRESS
EMPLOYEE ---> AGE
EMPLOYEE --->> CHILD

| R | | | |
|-----------------|-------------------|------------|--------------|
| EMPLOYEE | ADDRESS | AGE | CHILD |
| Adams | 3 N. F Street | 48 | Paul |
| Adams | 3 N. F Street | 48 | Jane |
| Adams | 3 N. F Street | 48 | Anne |
| Adams | 3 N. F Street | 48 | Fred |
| Baker | 22 S. M Street | 24 | |
| Chase | 17 E. 2nd. Avenue | 37 | Alan |
| Davis | 29 W. 5th Avenue | 50 | Mary |

Note: _____

The ADDRESS and AGE information for Adams must appear four times because Adams has four children. The DesignManager normalization procedure eliminates this sort of redundancy and the consequent need for multiple updating of records if, say, Adams moves to another address or has another child.

In generating a third normal form schema, DesignManager would instead construct two relations:

R1 (EMPLOYEE , ADDRESS , AGE)
R2 (EMPLOYEE , CHILD)

with instances indicated by these tables:

| R1 | | |
|----------|-------------------|-----|
| EMPLOYEE | ADDRESS | AGE |
| Adams | 3 N. F Street | 48 |
| Baker | 22 S. M Street | 24 |
| Chase | 17 E. 2nd. Avenue | 37 |
| Davis | 29 W. 5th Avenue | 50 |

| R2 | |
|-------|------|
| Adams | Paul |
| Adams | Jane |
| Adams | Anne |
| Adams | Fred |
| Chase | Alan |
| Davis | Mary |

Note:

In the R2 table, there is no need for a Baker entry.

Every relation of a relational database model must have a *key* to provide access to the values of its data elements in a physical implementation. A key of a relation is a minimal set of one or more of its data elements needed to identify uniquely all the values in each row of the relation when viewed as a table. That is, it is a minimal set of data elements that functionally determines all the data elements of the relation. In this relation, for example, each data element functionally depends on the key EMPLOYEE:

R1 (EMPLOYEE, ADDRESS, AGE)

However, EMPLOYEE is not the key to either of these two relations:

R2 (EMPLOYEE, CHILD)

R (EMPLOYEE, ADDRESS, AGE, CHILD)

Since no subset of the data elements of R2 functionally determines all of its data elements (CHILD is multiply determined by EMPLOYEE and no dependency whatever is specified from CHILD to EMPLOYEE), it must be true that the entire set of data elements is the key. Such a relation is said to be *all-key* (it is also referred to as an *MVD-relation*). In relation R, the minimal set of data elements that functionally determines all of its data elements again is EMPLOYEE + CHILD. Consider these FDs:

```
EMPLOYEE + CHILD ---> ADDRESS
EMPLOYEE + CHILD ---> AGE
```

The above FDs follow directly from these specified FDs, respectively:

```
EMPLOYEE ---> ADDRESS  EMPLOYEE ---> AGE
```

Note: _____

The relation R is not even in second normal form because the data elements ADDRESS and AGE are not *fully functionally dependent* on the key, being functionally dependent on one of the key components, EMPLOYEE. Relations R1 and R2, on the other hand, are both in third normal form.

The primary purpose of a logical database model is to ensure that users of the model are able to access values of the data elements required for a particular application in accordance with their view of the associations (dependencies in a relational model). This requirement is always satisfied in a relational schema produced by DesignManager, in that each dependency MERGED into the workbench design area and input to the design procedure is represented in the resulting schema either directly by a single relation or indirectly via several relations. For example, consider this FD:

```
EMPLOYEE ---> AGE
```

The FD specified above is represented directly by this relation where EMPLOYEE is the key:

```
R1 (EMPLOYEE, ADDRESS, AGE)
```

On the other hand, consider these FDs:

```
EMPLOYEE ---> GRADE
GRADE ---> SALARY-SCALE
EMPLOYEE ---> SALARY-SCALE
```

The third FD shown in the preceding example would be removed as *transitively* redundant and would be represented indirectly in the 3NF relations generated by the DesignManager design procedure:

```
R1 (EMPLOYEE, GRADE)
R2 (GRADE, SALARY-SCALE)
```

An Example Illustrating Normalization Benefits

This example further highlights some of the advantages of the DesignManager normalization procedure. An unnormalized arrangement of data describing the suppliers of an organization, the cities in which they are based, and the code number assigned to each city, is shown in the following table, an instance of the relation:

R (SUPPLIER, CODE, CITY)

(with SUPPLIER as the key) based on the FDs:

SUPPLIER ---> CITY
 SUPPLIER ---> CODE
 CITY ---> CODE

| R | | |
|----------|---------|------|
| SUPPLIER | CITY | CODE |
| Suppl-1 | London | 20 |
| Suppl-2 | London | 20 |
| Suppl-3 | Sydney | 30 |
| Suppl-4 | Sydney | 30 |
| Suppl-5 | Sydney | 30 |
| Suppl-6 | Seattle | 10 |
| Suppl-7 | Hamburg | 40 |
| Suppl-8 | Sydney | 30 |
| Suppl-9 | Seattle | 10 |
| Suppl-10 | London | 20 |

Such an arrangement presents a number of problems when a user attempts to modify some of the data values. Consider, for example, these problems which could arise in adding, deleting, and updating values in the table:

Inserting. Information on the code of a new city cannot be stored unless a supplier exists there.

Deleting. If Suppl-7 goes out of business or is no longer used as a supplier and thus is removed from the table, the information regarding the code number of Hamburg will be lost.

Updating. If the code number of a city changes, multiple records must be altered because the code number entry (except for Hamburg) is held redundantly.

The DesignManager design procedure would first normalize the given dependencies by removing the FD as transitively redundant:

SUPPLIER ---> CODE

Then, instead of the single relation R shown above, it would generate the 3NF relations with SUPPLIER and CITY as the respective keys:

R1 (SUPPLIER, CITY) R2 (CITY, CODE)

These tables would result, which are free of the above difficulties:

| R1 | |
|-----------------|-------------|
| SUPPLIER | CITY |
| Suppl-1 | London |
| Suppl-2 | London |
| Suppl-3 | Sydney |
| Suppl-4 | Sydney |
| Suppl-5 | Sydney |
| Suppl-6 | Seattle |
| Suppl-7 | Hamburg |
| Suppl-8 | Sydney |
| Suppl-9 | Seattle |
| Suppl-10 | London |

| R2 | |
|-------------|-------------|
| CITY | CODE |
| London | 20 |
| Sydney | 30 |
| Seattle | 10 |
| Hamburg | 40 |

Note: _____

Apart from the redundancy in CITY values deliberately imposed in order to eliminate updating problems, the tables which result from the normalized relations incur the minimum redundancy of data values.

The DesignManager Design Procedure

The DesignManager design procedure is initiated by the DESIGN command. The procedure consists of two major functions, normalization of the dependencies held as a composite view in the workbench design area, and generation of 3NF relations. These steps are required (perform ["Step 1: Elimination of Extraneous Data Elements" on page 139](#) to ensure that the resulting relational schema is in second normal form and ["Step 2: Removal of Redundant Dependencies" on page 140](#) to ensure that the schema is in third normal form):

Step 1: Elimination of Extraneous Data Elements

Each FD in the composite view which contains two or more data elements on its left-hand side is examined to see if any of these data elements may be eliminated from the left-hand side as extraneous. This is done, for a given FD, by testing the FD obtained from it to see if it can be derived from all the FDs in the composite view (including the given one). For example, if this FD appears in the composite view:

$$A + B \text{ ---} > C$$

Then a test first is made to see if this FD obtained by eliminating the data element A from the left-hand side of the given FD is implied by all the FDs (including the given one):

$$B \text{ ---} > C$$

If so, then this FD:

$$A + B \text{ ---} > C$$

is equivalent to this FD in the composite view and is replaced by it:

$$B \text{ ---} > C$$

Otherwise, a second test is made to see if B is extraneous in the given FD. In particular, consider these examples:

- If the FDs:

$$\text{EMPL-NO} \text{ ---} > \text{GRADE}$$

$$\text{EMPL-NO} + \text{PROJ-NO} \text{ ---} > \text{GRADE}$$

both appear in the composite view (probably from different userviews), then PROJ-NO is extraneous in the second FD and the DesignManager software eliminates it (any value of PROJ-NO with a given value of EMPL-NO would determine the same value of GRADE).

- If the FDs:

$$\text{EMPL-NO} \text{ ---} > \text{GRADE}$$

$$\text{EMPL-NO} + \text{GRADE} \text{ ---} > \text{SAL-SCALE}$$

are given, then GRADE is extraneous in the second FD and is eliminated (the access requirements of the original FD are still satisfied since a given value of EMPL-NO determines a value both for GRADE and for SAL-SCALE).

Step 2: Removal of Redundant Dependencies

Each FD (in the composite view after ["Step 1: Elimination of Extraneous Data Elements" on page 139](#) is completed) is tested in turn to see if it is implied by the rest of the FDs in the composite view. If so, its access requirements can be satisfied by one or more other FDs, and the FD under test is removed from the composite view as redundant. For example, if these FDs:

A ---> B

B ---> C

A ---> C

all appear in the composite view, A ---> C is removed as (transitively) redundant. Similarly, if all of these appear:

A ---> B

B + D ---> C

A + D ---> C

then A + D ---> C is removed. A particular example is given in ["An Example Illustrating Normalization Benefits" on page 137](#) above, where the FD shown here is transitive in the given set of FDs and is removed as redundant:

SUPPLIER ---> CODE

After all FDs are tested for redundancy, each MVD in the composite view is tested to see whether or not it is derivable (as if it were an FD) from the remaining set of FDs. If it is, this means that its access requirements are satisfied by the FDs alone and that an inconsistency is present in the composite view. Either the MVD should have been entered (to the modeling dictionary in a USERVIEW) as an FD or there is a mistake in one or more of the FDs. As a consequence, the MVD is removed from the composite view as inconsistent, an inconsistency flag is set for the composite view (see [remark 6 on page 81](#) of the MERGE command for discussion of inconsistency flags and how they can be unset), and a warning message is given, that inconsistencies were found during the design process.

Step 3: Partitioning of the Composite View

As a preliminary to forming relations, the FDs remaining in the composite view after completion of ["Step 2: Removal of Redundant Dependencies" on page 140](#), separate into groups (called *FD-groups*) based on common left-hand sides (the potential keys of the FD-relations). Each MVD is placed in a group (called an *MVD-group*) of its own as a candidate for use in generating a separate all-key relation composed of all the data elements in the MVD (see ["Step 4: Merging Groups with Identical Potential Keys" on page 141](#) for exceptions and possible merging of certain MVD-groups with other MVD-groups and/or FD-groups possessing the same potential key). For example, this group of FDs could be formed from a composite view with EMPL-NO as the common left-hand side:

EMPL-NO ---> ADDR

EMPL-NO ---> GRADE

(assuming that there are no other FDs in the composite view with EMPL-NO as the only data element on the left-hand side). On the other hand, the FD:

EMPL-NO + SAL-DATE ---> SALARY

would not be contained in this group; instead, it would appear in a different group (possibly by itself) with EMPL-NO + SAL-DATE as the common left-hand side. Furthermore, the MVD:

EMPL-NO --->> CHILD

if present in the composite view, would not be included in either of the above groupings since it would be used to form a separate relation with the key EMPL-NO + CHILD (unless merged in ["Step 4: Merging Groups with Identical Potential Keys" on page 141](#) with another FD- or MVD-group having the same potential key). At the conclusion of ["Step 3: Partitioning of the Composite View" on page 141](#)) each group of FDs formed is the basis for a separate relation with the common left-hand side designated as the key, and each MVD-group is potentially the basis for a separate all-key relation composed of all the data elements in the MVD.

Step 4: Merging Groups with Identical Potential Keys

Although each group formed in ["Step 3: Partitioning of the Composite View" on page 141](#) (with the possible exception of certain MVD-groups such as group G4 in the example below) can be used to generate a 3NF relation, it is possible that the same data element or set of data elements is potentially the key of more than one relation, that is, two or more groups may have the same potential key. This is particularly undesirable after schema generation when naming relations via the KEY option in the NAME command. Thus, if two or more groups have a potential key in common, they are merged in the workbench design area into a single group with the same potential key and containing all the dependencies from all the separate groups. For example, if these groups of dependencies were formed in the workbench design area, each with the potential key A + B:

G1 : A + B ---> C, A + B ---> D

G2 : A --->> B

G3 : B --->> A

they would be merged into the single group:

G: A + B ----> C, A + B ----> D, A ---->> B, B ---->> A

The resulting group G would be used (in ["Step 5: Generation of 3NF Relations" on page 143](#)) to construct a single relation:

R (A, B, C, D)

keyed by A + B and representing all four of the dependencies in the group. A more subtle situation requiring the merging of groups arises when an MVD-group forms whose potential key does not consist of all the data elements in the MVD (recall that each MVD-group is formed from a single MVD; see ["Step 3: Partitioning of the Composite View" on page 141](#)). This occurs if an FD appears in one of the FD-groups (or is expressible by the union of several FDs from the same FD-group) which contains exactly the same data elements as the MVD. To illustrate, suppose that, in the example above, this group was also present in the workbench design area:

G4: C + B ---->> A

If group G4 is considered in isolation from any of the other groups in the workbench design area, one might assume that the potential key consists of all the data elements in the group (as in the case of groups G2 and G3 above) and so that group G4 could be used to generate an all-key relation R(C,B,A) containing the data elements A, B, and C. However, due to the presence of this FD in group G1 above (which is composed of exactly the same data elements:

A + B ----> C

it follows that such a relation could not be all-key. Instead, the key would be A + B. (Recall that the key of a relation must be a *minimal* set of data elements which functionally determines every data element in the relation; see ["Relations and Keys" on page 134](#) and ["Glossary" on page 245](#).) Thus the potential key of group G4 is A + B and, as a consequence, group G4 merges with groups G1, G2, and G3; that is, this MVD would be added to the group G above:

C + B ---->> A

The resulting relation again keyed by A + B, represents all five of the dependencies in the group:

R (A, B, C, D)

For a simpler illustration of an MVD which cannot be used to generate an all-key relation, consider these dependencies, perhaps arising in a personnel application:

EMPL-NO ----> DEPT-NO

DEPT-NO ---->> EMPL-NO

In "[Step 3: Partitioning of the Composite View](#)" on page 141, the first dependency would be placed in a group containing every FD in the workbench design area having EMPL-NO (the potential key) as the left-hand side, and the MVD becomes the sole member of an MVD-group, as this:

G1 : EMPL-NO ---> DEPT-NO, . . .

G2 : DEPT-NO --->> EMPL-NO

The potential key of G2 must be the single data element EMPL-NO rather than DEPT-NO + EMPL-NO because EMPL-NO functionally determines both DEPT-NO and EMPL-NO (the latter trivially). Thus G2 must be merged with G1 to produce a single group:

G : EMPL-NO ---> DEPT-NO, . . . , DEPT-NO --->> EMPL-NO

which is used (in "[Step 5: Generation of 3NF Relations](#)" on page 143) to generate a single relation having EMPL-NO as the key and representing all the dependencies in G, including DEPT-NO --->> EMPL-NO:

R (EMPL-NO, . . . , DEPT-NO)

In both examples above, only one of the groups contains FDs. The others are all MVD-groups. It is always true, when a set of groups is to be merged on a common potential key, that either one or none of the groups in the set can contain FDs. This is due to the method used for grouping FDs in "[Step 3: Partitioning of the Composite View](#)" on page 141. A group can be formed by merging only MVD-groups if no FD-group appears in the workbench design area having the same potential key. In this case, each group merged contains exactly the same data elements. Consider, for example, groups G2 and G3 in the first illustration above if group G1 is omitted, that is, if no FD having A + B as its left-hand side is present in the workbench design area.

In this case, group G4 would not be merged with G2 and G3. This step ensures that, in the 3NF schema generated in "[Step 5: Generation of 3NF Relations](#)" on page 143, no key identifies more than one relation.

Step 5: Generation of 3NF Relations

For each group of dependencies present in the workbench design area after completion of "[Step 3: Partitioning of the Composite View](#)" on page 141 and "[Step 4: Merging Groups with Identical Potential Keys](#)" on page 141, a relation is constructed containing every data element appearing in any dependency of the group. For any group containing one or more FDs, the data element or set of data elements comprising the common left-hand side of the FDs is identified as the key of the relation (even if the group contains MVDs; see "[Step 4: Merging Groups with Identical Potential Keys](#)" on page 141). Relations formed in this manner are *FD-relations*. The first group of FDs given in the example of "[Step 3: Partitioning of the Composite View](#)" on page 141 would be used to generate this relation with EMPL-NO designated as the key:

R (EMPL-NO, ADDR, GRADE)

For each MVD-group present in the workbench design area, a separate all-key relation is generated containing all the data elements appearing in the group (recall that, if two or more MVDs are contained in the group due to merging in ["Step 4: Merging Groups with Identical Potential Keys" on page 141](#), each is composed of exactly the same set of data elements). As indicated in ["Relations and Keys" on page 134](#), the only possible key of such a relation is the entire set of data elements and the relation is termed *all-key*. All-key relations are also referred to as *MVD-relations*. The separation of MVDs from FD relations (except when the keys are identical; see ["Step 4: Merging Groups with Identical Potential Keys" on page 141](#)) is necessary to ensure that the relations are in third normal form (or even second normal form; see the example involving EMPLOYEE, ADDRESS, AGE, and CHILD in ["An Example Illustrating Normalization Benefits" on page 137](#)).

Naming Relations and Storing Them in the Dictionary

After relations are generated using the DESIGN command, the user can name the relations in the workbench design area by means of one or more NAME commands (see ["Design Mode Command Specifications" on page 73](#)). This section describes methods for carrying out the naming procedure and subsequently storing the relations in the modeling dictionary (as USERVIEW members).

To begin with, it must be kept in mind that any previously generated relations are not preserved in the workbench design area when a new DESIGN command processes. That is, a second DESIGN command will not be accepted unless preceded by at least one intervening MERGE command, and part of the action of the MERGE command is to remove all relations which are present in the workbench design area. All previously generated relations are lost along with their names. The procedure given below for naming relations is therefore recommended when iterating a design to avert the tedious process of coding the NAME command(s) all over again, particularly when most of the newly generated relations are identical to relations which have already been generated and NAMED. It is much simpler instead to preserve a fixed sequence of one or more NAME commands and re-enter the sequence each time the design (or a subset of the design) is iterated. The fixed sequence may require additions to cater for both name changes and for relations generated for the first time in subsequent iterations, but most of the relations are ones already generated and therefore having a corresponding NAME command in the sequence (see the example of producing such a sequence at the end of this section). In conjunction with the naming procedure, a method is outlined for storing the generated relations in the modeling dictionary as USERVIEW members.

The following procedure for saving a sequence of NAME commands and storing relations in the dictionary is available to users in Integrated DesignManager/DataManager installations. Modifications required for users in non-integrated installations are discussed concurrently. The procedure, as required for integrated users, is presented in [Figure 5 on page 148](#). It may be useful to follow this diagram while reading the discussion below: requirements that differ for non-integrated users are obvious from the discussion.

The procedure consists of these steps:

1. Perform the First DesignManager Run (see [Figure 5 on page 148](#))

Perform the first DesignManager run either interactively or in batch. Include a MERGE command for all the userviews associated with the design, a DESIGN command (preferably specifying the keywords [WITH] AUDIT DETAILS), a SWITCH OUTPUT command to direct all output to a file which can be saved for subsequent modification and editing. Additionally, include a LIST RELATIONS command to determine the key of each relation generated, a REPORT LOGICAL-SCHEMA DETAILS command to permit in-depth analysis of the relations, and a STORE command to save the content of the workbench design area for a subsequent run (see [step 4](#)) in which the relations are named. (A PLOT LOGICAL-SCHEMA command can also be entered if the optional additional facility User Printer Graphics is installed.) For non-integrated users, this step must be performed in batch and output is directed to the file via the job control instead of SWITCH OUTPUT.

2. Edit the Output and Convert the Listing (see [Figure 5 on page 148](#))

By editing the output of the LIST RELATIONS command in the file generated in [step 1](#), the user can convert the listing to a sequence of one or more NAME commands which can be preserved for use in subsequent design iterations. An example is given below illustrating an edited output file. This step is the same for both integrated and non-integrated users.

3. Run Batch to ADD Sequence of Commands to the Modeling Dictionary (see [Figure 5 on page 148](#))

A DesignManager run is performed in batch to ADD the sequence of NAME commands generated in [step 2](#) to the modeling dictionary as a COMMAND-STREAM member. No design mode commands are required. This step is not available to non-integrated users.

4. Name Relations and Set up another Output File (see [Figure 5 on page 148](#))

A run is performed interactively or in batch to name the relations and to set up another output file for editing so that the relations can be subsequently ADDED (see [step 6 on page 147](#)) to the modeling dictionary, including:

- A FETCH command to retrieve the content of the workbench design area STORED in [step 1](#).
- A PERFORM command to name the relations (using the COMMAND-STREAM member defined in [step 3](#)).
- A SWITCH OUTPUT command to save output for subsequent editing.
- A REPORT LOGICAL-SCHEMA DETAILS command, to provide output for subsequent editing (see [step 5 on page 146](#)), and to permit analysis of the now named relations.

- If desired, a STORE command to save the content of the workbench design area to be FETCHed and augmented via the MERGE command in a subsequent design iteration.

For non-integrated users, perform this step in batch with these modifications:

- The sequence of NAME commands is input (via the job control) directly from the output file edited in [step 2 on page 145](#) rather than from a COMMAND-STREAM member.
- Output is directed to a new output file via the job control rather than a SWITCH OUTPUT command.

5. Edit Output and Convert to a Sequence of Data Definitions (see [Figure 5 on page 148](#))

By editing the output of the REPORT LOGICAL-SCHEMA DETAILS command in the output file produced in [step 4 on page 145](#), the user can convert it (for subsequent ADDing to the modeling dictionary, see [step 6 on page 147](#)) to a sequence of USERVIEW data definitions, one per generated relation, each preceded by an ADD command specifying the corresponding relation name. The output for each relation in the report ends with a GENERATED SYNTAX section containing a list of the dependencies represented by the relation, appearing in the form of USERVIEW member definition syntax, preceded by an ADD command identifier and the relation name. All that is required by the user is to remove the rest of the REPORT LOGICAL-SCHEMA DETAILS output, and then, in the GENERATED SYNTAX section, complete the data definition for each relation with:

```
CATALOGUE 'RELATION'
```

This use of the catalogue 'relation' is to distinguish in the modeling dictionary between USERVIEW members that represent named relations generated from the composite view in the workbench design area, and those stored to represent a particular user's application-oriented access requirements.

The output for a relation EMPLOYEE is edited to read:

```
ADD EMPLOYEE .
USE RVIEW
DEP LHS  EMPL-NO
      FD RHS                               EMPL-NAME
                                           , ADDRESS
                                           , DEPT-NO

CATALOGUE 'RELATION'
```

(If the optional additional facility User Formatted Output is available, the user can set up a format member to output the generated syntax directly in the form shown above.) This step is the same for both integrated and non-integrated users.

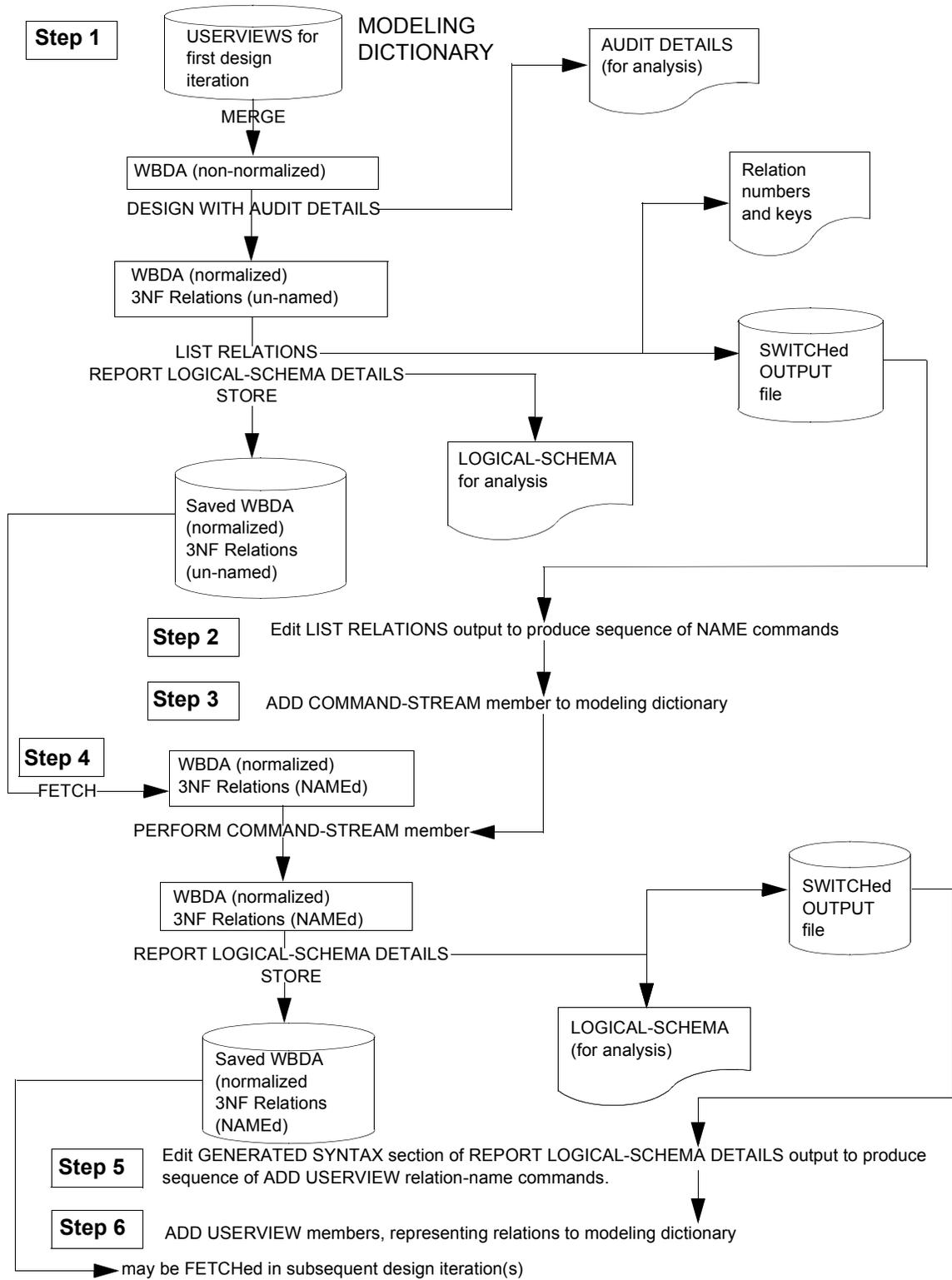
6. ADD Relations to the Modeling Dictionary as USERVIEW Members (see [Figure 5 on page 148](#))

A DesignManager run is performed in batch to ADD the relations (generated in [step 4 on page 145](#)) to the modeling dictionary as USERVIEW members. The appropriate sequence of ADD commands and USERVIEW data definitions are input via the job control directly from the output file edited in [step 5 on page 146](#). This step is the same for both integrated and non-integrated users.

[Step 1 on page 145](#) through [step 6](#) constitute the procedure for NAMEing relations and ADDing them as USERVIEWs in the first design iteration. For subsequent design iterations, the content of the workbench design area (STOREd in a previous iteration) can be FETCHed and augmented via the MERGE command to provide input for the new iteration. The COMMAND-STREAM member defined in [step 3 on page 145](#) of the first iteration can be used to NAME the relations. Using the dictionary mode MODIFY command, further NAME commands can be added to this member to cater for both relation name changes and newly generated relations. Non-integrated users would NAME relations directly from the output file edited in [step 2 on page 145](#) of the first design iteration, perhaps re-editing to add NAME commands catering for relation name changes and newly generated relations. An initial re-editing of this file could also be used to preface the first NAME command with FETCH, MERGE, DESIGN, LIST, and REPORT commands, as required, to begin a new iteration.

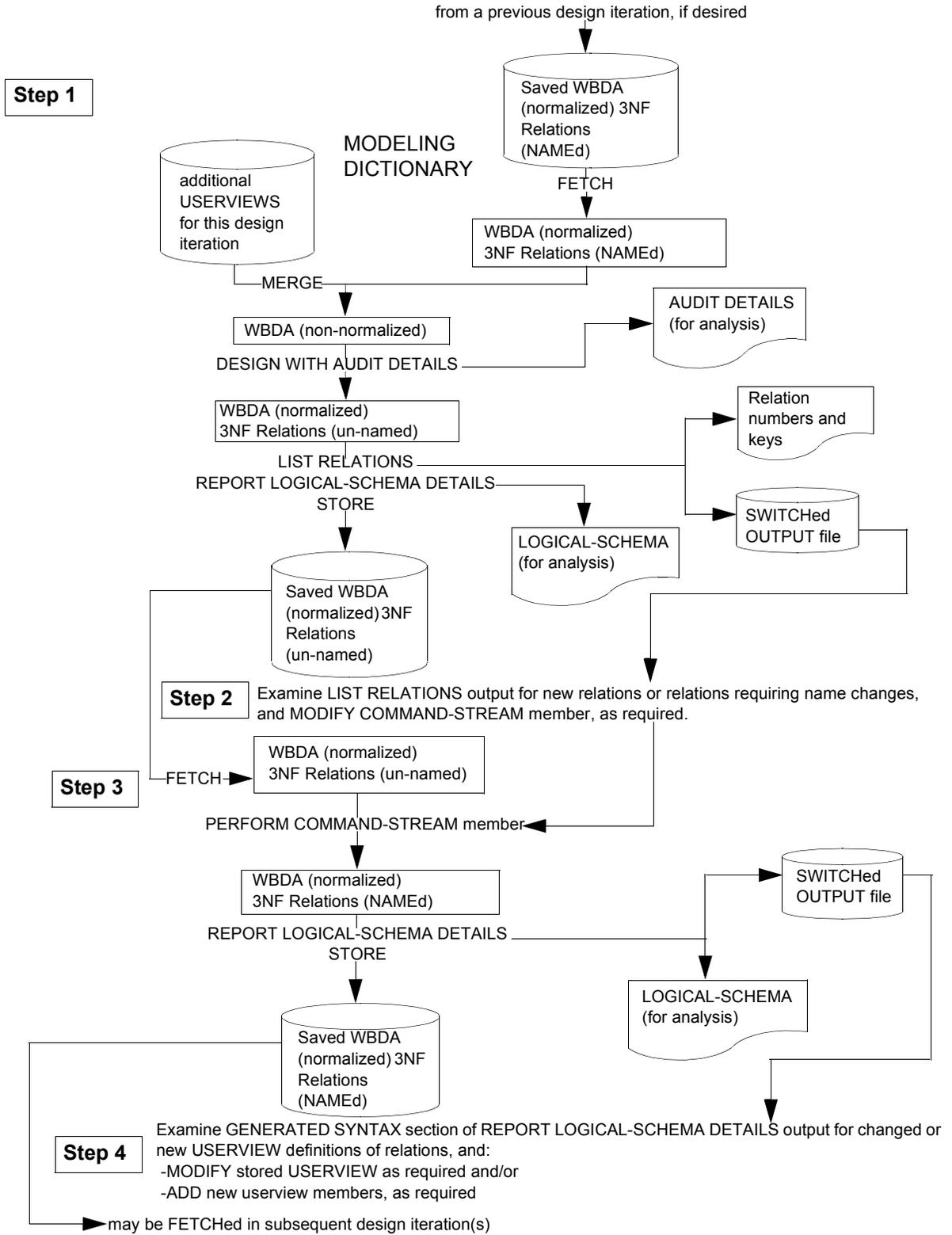
At the end of each subsequent iteration, the dictionary mode ADD and MODIFY commands can be used to store as USERVIEW members any new relations generated, and to update existing USERVIEW data definitions for any relations that have changed. The steps required for integrated users to perform a subsequent design iteration are summarized in [Figure 6 on page 149](#). The requirements that differ for non-integrated users correspond to those highlighted in the discussion of steps for the first design iteration.

Figure 5. First Design Iteration, to Generate Relations, Name them, and Store them in the modeling Dictionary (for Integrated Environments)



WBDA = Workbench Design Area

Figure 6. Subsequent Design Iteration (for Integrated Environments)



WBDA = Workbench Design Area

This example illustrates the procedure indicated in [step 2 on page 145](#) for producing one or more NAME commands from the output of the LIST RELATIONS command. Suppose this output is produced by the LIST command in the first design iteration:

LIST OF RELATIONS HELD IN WORKBENCH DESIGN AREA

| NUMBER | KEY | NAME |
|--------|----------------------|-------------|
| 1 | DIV-NO | **NO NAME** |
| 2 | DEPT-NO | **NO NAME** |
| 3 | EMP-NO | **NO NAME** |
| 4 | EMP-NO CHILD-NAME | **NO NAME** |
| 5 | GRADE | **NO NAME** |
| 6 | EMP-NO SAL-DATE | **NO NAME** |

A relation number and key characterizes each relation in the listing; relation names, however, have not yet been assigned. Suppose further that, based on an analysis of all the output produced, the user selects these names, ordered by relation number, for the relations generated: DIVISION, DEPARTMENT, EMPLOYEE, CHILDREN, SALARY-STRUCTURE, and SALARY-HISTORY. The user would then make a copy of the output from the print file, delete everything except the listing shown above, and edit the listing to conform to the NAME command syntax replacing ** NO NAME ** in each instance with the desired relation name. The edited output should then appear in this (or an equivalent) form:

| NAME | RELATIONS | | |
|------|-----------------------|----|------------------|
| KEY | DIV-NO | AS | DIVISION |
| KEY | DEPT-NO | AS | DEPARTMENT |
| KEY | EMPLOYEE | AS | EMPLOYEE |
| KEY | EMP-NO, CHILD-NAME | AS | CHILDREN |
| KEY | GRADE | AS | SALARY-STRUCTURE |
| KEY | EMP-NO, SAL-DATE | AS | SALARY-HISTORY; |

The key rather than the relation number was used in the NAME command to identify each relation. (Recall that no two relations can have the same key; see ["Step 5: Generation of 3NF Relations" on page 143](#) of the design procedure in ["The DesignManager Design Procedure" on page 138](#).) This practice is recommended whenever possible, particularly when generating many relations, because in subsequent iterations some relations may be changed or removed and new ones added.

Exercising the KEY rather than the NUMBER option usually enables the original form of the NAME command to remain fixed for these reasons:

- If a relation doesn't change (or changes only in its non-key data elements), the key remains the same, whereas the relation number is very likely to vary.
- If in a subsequent iteration there is no relation in the WBDA with the key specified in a KEY clause of the NAME command, no harm is done; the software outputs a warning message to that effect and go on processing.
- If the user wants to rename a relation with a given key, they can do it by adding a new NAME command to the file; when the new command is processed, the old name specified in the original command is overwritten by the name specified in the new command.

After subsequent design iterations, new NAME commands may be added to this list, as required.

Use of Pseudo-FDs to Handle Subcategories

In addition to functional and multivalued dependencies, DesignManager permits specification of *pseudo-FDs* in order to define subcategories of data elements. A pseudo-FD from data element A to data element B, as shown here, states that A is a subcategory of B and takes on values from a subset of the domain of B:

A ==> B

Syntactically, a pseudo-FD is handled by the DesignManager software just as any other FD. Semantically, it states the trivial condition that any value of A *is* a value of B (viewed as a table of values, the A and B columns would be identical). Handled correctly, the pseudo-FD is a powerful tool which can be used to avoid many naming and dependency anomalies; in particular, it can be used to solve the parts explosion [or Bill of Materials Processor (BOMP)] problem, in which a user wishes to find subparts of a part which themselves are parts capable of having subparts.

These are examples of pseudo-FDs which could be defined for the indicated areas of application:

- For a stock exchange application:

```
BUYER-FIRM ===> MEMBER-FIRM  
SELLER-FIRM ===> MEMBER-FIRM
```

- In a university environment:

```
ADVISING-PROFESSOR ===> PROFESSOR  
EXAMINING-PROFESSOR ===> PROFESSOR
```

- For use in a *parts explosion* listing all component parts of a part:

```
MAJOR-PART ===> PART  
MINOR-PART ===> PART
```

- For a military logistics inventory:

```
LAND-VEHICLE ===> VEHICLE  
SEA-VERICLE ===> VEHICLE  
AIR-VEHICLE ===> VEHICLE
```

The last example differs significantly from the first three in that it illustrates a set of non-overlapping (that is, mutually exclusive) subcategories, which usually can be handled without difficulty. They are specified to denote different objects in a generic hierarchy and are not likely to be involved in naming or dependency anomalies. In the first three examples, on the other hand, the subcategories are overlapping (for instance, a BUYER-FIRM in one transaction can be a SELLER-FIRM in another), and in some instances may overlap completely (each BUYER-FIRM is at some point a SELLER-FIRM and vice versa) or almost completely. In these examples, the left-hand side of each pseudo-FD can be perceived as indicating a *role* played by the right-hand side. (Similarly, a relation generated from such a pseudo-FD is termed a *role* relation.) The role may be completely dynamic as in the stock exchange or parts explosion examples, or it may be somewhat static as in the case of ADVISING-PROFESSOR. In either event, a role application is usually characterized by the need for one or more additional dependencies to define the context of the role, such as:

```
STUDENT ---> ADVISING-PROFESSOR
```

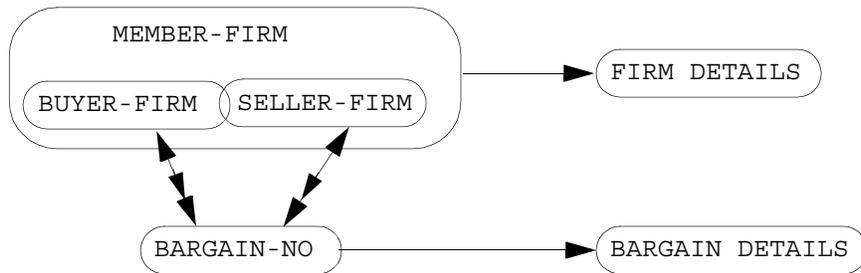
or

```
MAJOR-PART --->> MINOR-PART
```

A LAND-VEHICLE, on the other hand, is simply a type of vehicle and is not a role, nor does it require additional dependencies to define its function as a subcategory.

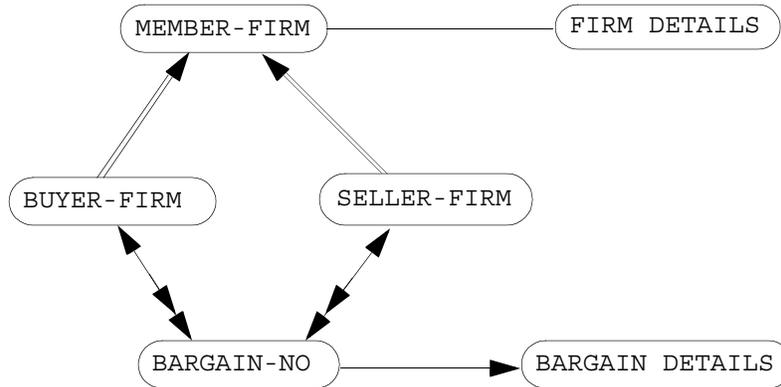
As indicated earlier, the use of pseudo-FDs can be helpful in many application areas. This is particularly true in the case of role situations. A number of examples follow illustrating some of the problems involved and how they can be resolved. It is often useful, in approaching such a problem, to view it first in terms of sets and subsets before defining pseudo-FDs.

Consider this stock exchange application, where each transaction involving the purchase of stock by one member firm from another is termed a bargain and is identified by a unique bargain number. Each bargain number determines a single buyer and a single seller amongst the member firms. Each firm is characterized by descriptive details (such as address) and each bargain by the details of the transaction. If BARGAIN-NO and MEMBER-FIRM are specified as data elements, then, to handle the anomaly of two distinct FDs holding from BARGAIN-NO to MEMBER-FIRM, it is necessary to define the data elements, BUYER-FIRM and SELLER-FIRM, as subcategories to indicate the roles played by the data element MEMBER-FIRM in each transaction. Representing the data elements as sets connected by arrows, the dependency information given above can be pictured in this form:



where BUYER-FIRM and SELLER-FIRM are overlapping subsets of MEMBER-FIRM. Bidirectional arrows connect BARGAIN-NO with BUYER-FIRM and SELLER-FIRM, indicating FDs in one direction (from BARGAIN-NO) and MVDs in the other.

To handle the subcategory problem and ensure, for each BARGAIN-NO, that FIRM-DETAILS can be obtained for the corresponding BUYER-FIRM and SELLER-FIRM, pseudo-FDs are inserted from the subsets to the containing set as shown in this diagram:



where the double-shafted arrows indicate pseudo-FDs.

In our usual userview notation, this would appear in this form:

```

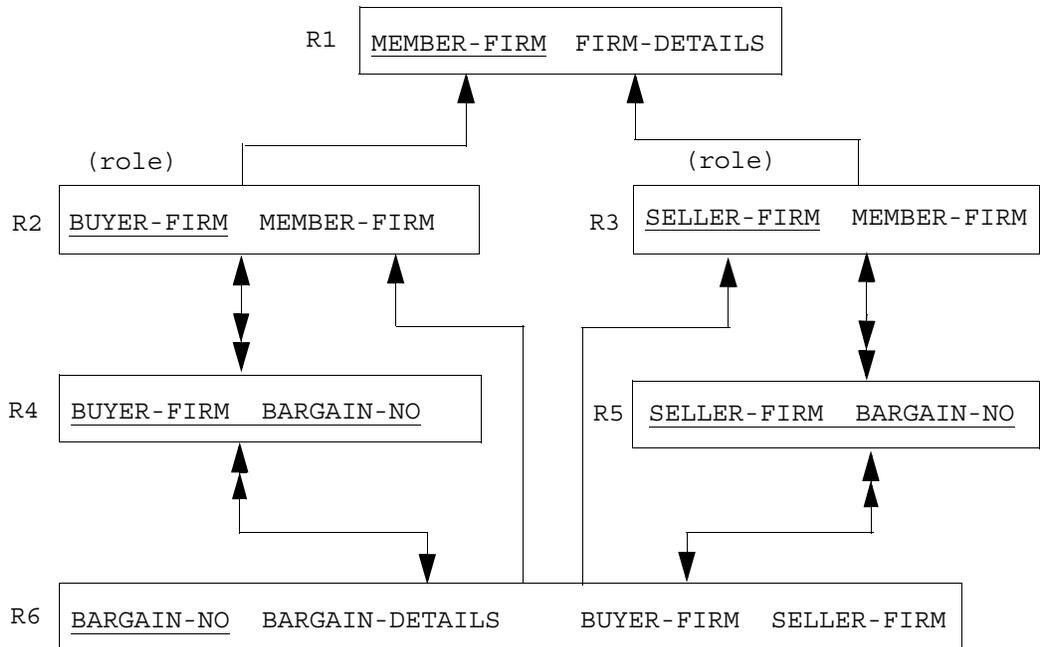
MEMBER-FIRM ---> FIRM-DETAILS
BUYER-FIRM ---> MEMBER-FIRM          (pseudo-FD)
BUYER-FIRM ---->> BARGAIN-NO
SELLER-FIRM ---> MEMBER-FIRM          (pseudo-FD)
SELLER-FIRM ---->> BARGAIN-NO
BARGAIN-NO ---> BUYER-FIRM
BARGAIN-NO ---> SELLER-FIRM
BARGAIN-NO ---> BARGAIN-DETAILS
    
```

In the userview, no distinction is made between the pseudo-FDs and the other FDs. (A COMMENT or NOTE clause could be inserted in the USERVIEW member, indicating the presence of the pseudo-FDs. The ITEM members defined for the data elements BUYER-FIRM and SELLER-FIRM should contain CATALOGUE clauses describing them as subcategories of MEMBER-FIRM.) The DesignManager software would treat them exactly as any other FD and would generate relations as these:

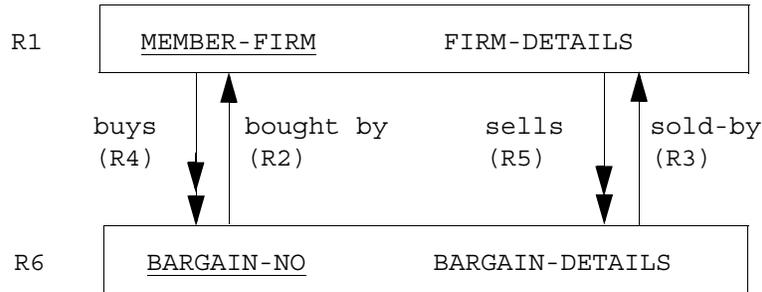
```

R1 (MEMBER-FIRM, FIRM-DETAILS)
R2 (BUYER-FIRM, MEMBER-FIRM)      (role relation)
R3 (SELLER-FIRM, MEMBER-FIRM)      (role relation)
R4 (BUYER-FIRM BARGAIN-NO)
R5 (SELLER-FIRM BARGAIN-NO)
R6 (BARGAIN-NO, BARGAIN-DETAILS, BUYER-FIRM, SELLER-FIRM)
    
```

where keys are underlined (R4 and R5 are all-key corresponding to MVDs), and foreign keys appear in R2, R3, R4, R5, and R6 (MEMBER-FIRM in R2 and R3, BARGAIN-NO in R4 and R5, and BUYER-FIRM and SELLER-FIRM in R6). Representing foreign key pointers by single-headed arrows and pointers to concatenated keys by double-headed arrows, the generated relations are depicted graphically in this form:



For a given BARGAIN-NO, we know the BUYER-FIRM and can obtain its FIRM-DETAILS in relation R1 through the role relation R2. The same is true for the SELLER-FIRM using the role relation R3. The functions performed in the relational schema shown above can be clarified through the interpretation given in this diagram, where R1 and R2 are connected by labeled arrows:



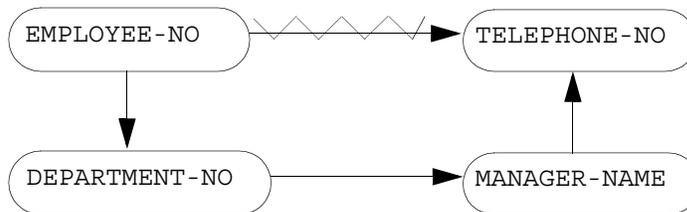
In the stock exchange example, subcategories and pseudo-FDs were introduced primarily to handle the problem of not being able to specify two different FDs from one data element (BARGAIN-NO) to another (MEMBER-FIRM). This is essentially a homonym problem. In the next example, the use of subcategories serves not only to avoid the occurrence of homonyms, but also, by clarifying the meaning of specified dependencies, to prevent removal of another dependency which is required during the normalization process. Consider these FDs:

EMPLOYEE-NO ---> TELEPHONE-NO
 MANAGER-NAME ---> TELEPHONE-NO

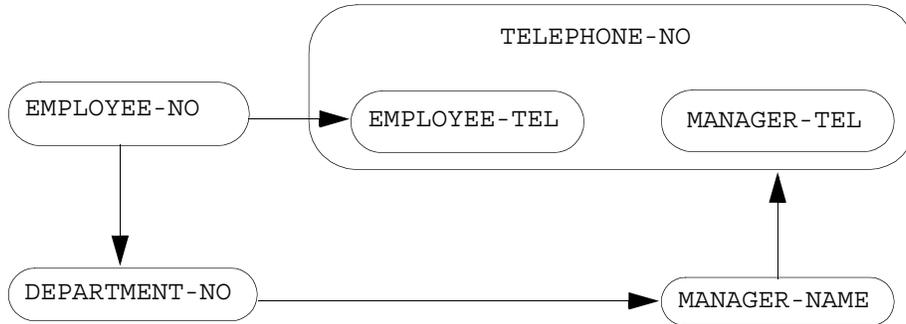
On the basis of the intersecting data element TELEPHONE-NO (see "[Merging Userviews into the Workbench Design Area](#)" on page 130), it is advisable to distinguish between employee's telephone numbers and manager's telephone numbers. If, however, these FDs are also specified:

EMPLOYEE-NO ---> DEPARTMENT-NO
 DEPARTMENT-NO ---> MANAGER-NAME

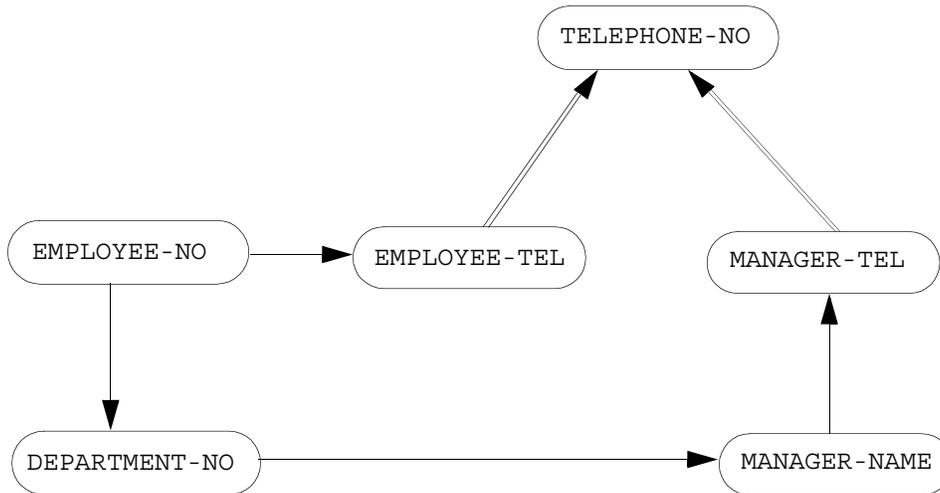
then the distinction becomes mandatory to prevent the removal of EMPLOYEE-NO ---> TELEPHONE-NO as transitively redundant, as shown by this diagram:



Going back to the set/subset notation of the last example, the subcategory data elements EMPLOYEE-TEL and MANAGER-TEL are introduced, as shown here:



or, using the pseudo-FD notation:



The transitivity is no longer present. The advantage of retaining the data element TELEPHONE-NO, and thus the pseudo-FDs along with the subcategories, becomes apparent if reference is made (perhaps in a subsequently analyzed userview) to the FD:

TELEPHONE-NO ---> OFFICE-NO

Because of the designated pseudo-FDs, access to OFFICE-NO from EMPLOYEE-NO and MANAGER-NAME is possible via EMPLOYEE-TEL and MANAGER-TEL, respectively.

Finally, the BOMP (Bill of Materials Processor) is a problem. The discussion is presented in terms of a parts explosion; however, many other examples of the BOMP problem can be proposed, such as determining the courses in a university curriculum which are prerequisite to a given course. In a parts explosion, a hierarchy of parts (for example, all the parts comprising a car, including the car itself) presents a concern. Each part in the hierarchy (except the basic components) is composed of subparts, which in turn are parts which may be composed of subparts, and so on. To determine the subparts of a given part from a list of all the parts in the hierarchy (represented, say, by the data element PART-NO) is a recursion problem requiring a many-valued mapping from the domain of PART-NO onto itself. It is important to recognize, however, that such a mapping is *not* given by the MVD:

PART-NO --->> PART-NO

which merely maps a given value of PART-NO into itself (and is in effect an FD), and which in any event would not be added to the workbench design area (WBDA) during a MERGE (see ["Merging Userviews into the Workbench Design Area" on page 130](#) on elimination of extraneous data elements from the right-hand side of a dependency). What is required is a way to map a given value from the domain of PART-NO onto a subset of the same domain which does not contain the given value. Thus, given the part P1, the corresponding set of subparts might be P5, P7, and P9, perhaps several of each. Once again, the problem is resolved by defining subcategories and pseudo-FDs.

In the illustration of a parts explosion, assume these requirements:

- For each part, find all of its (immediate) subparts and, for each subpart, the quantity used.
- For each part and each subpart determined, find its price and description.

The second requirement presents no problem. It is satisfied by defining data elements PART-NO, PRICE, and DESCR to represent a part, its price, and its description, respectively, and specifying the FD:

PART-NO ---> PRICE+DESCR

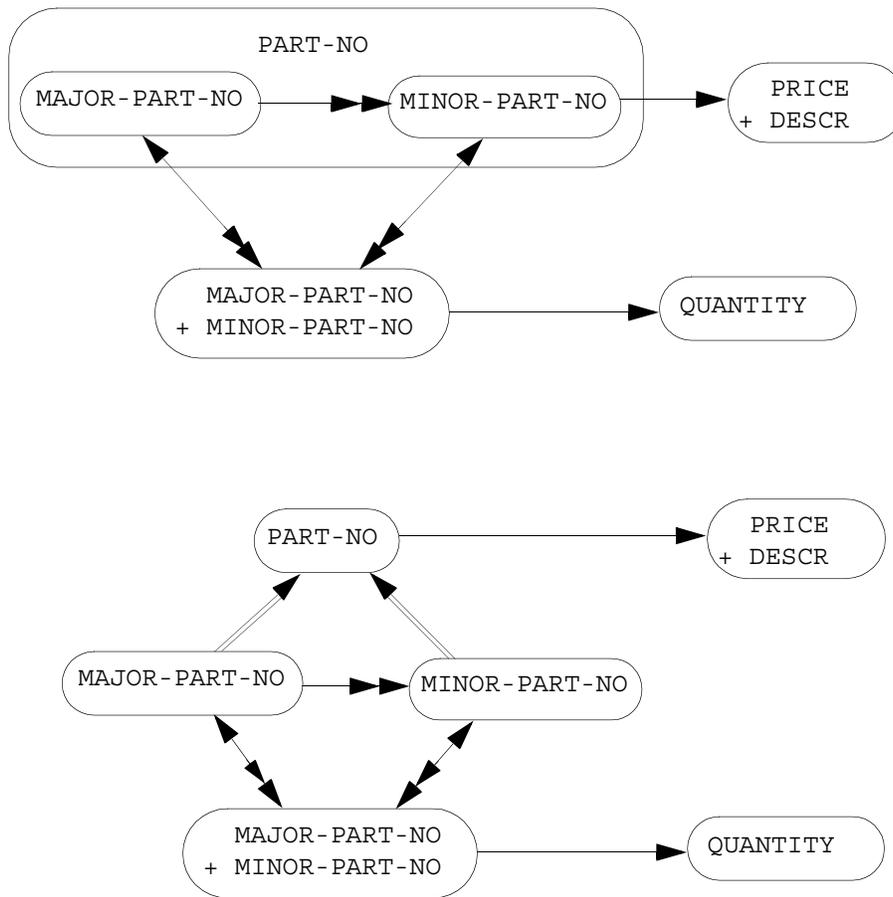
To satisfy the first requirement, however, distinguish between a part and its subparts. This can be done, as in the earlier examples, by defining subcategories and pseudo-FDs. Let the data elements MAJOR-PART-NO and MINOR-PART-NO represent a given part and one of its subparts, respectively, and the data element QUANTITY denote the number of instances of the subpart used in the given part. These dependencies are then specified, satisfying the first requirement:

MAJOR-PART-NO --->> MINOR-PART-NO
 MAJOR-PART-NO+MINOR-PART-NO ---> QUANTITY

To define MAJOR-PART-NO and MINOR-PART-NO as subcategories of PART-NO and permit use of the subcategories to access values of PRICE and DESCR, these pseudo-FDs must also be specified:

MAJOR-PART-NO ====> PART-NO
 MINOR-PART-NO ====> PART-NO

These diagrams represent this:



This would be interpreted in a userview as:

```

PART-NO          --->  PRICE+DESCR
MAJOR-PART-NO    --->  PART-NO          (pseudo-FD)
MINOR-PART-NO    --->  PART-NO          (pseudo-FD)
MAJOR-PART-NO    --->> MINOR-PART-NO
MAJOR-PART-NO+MINOR-PART-NO ---> QUANTITY
    
```

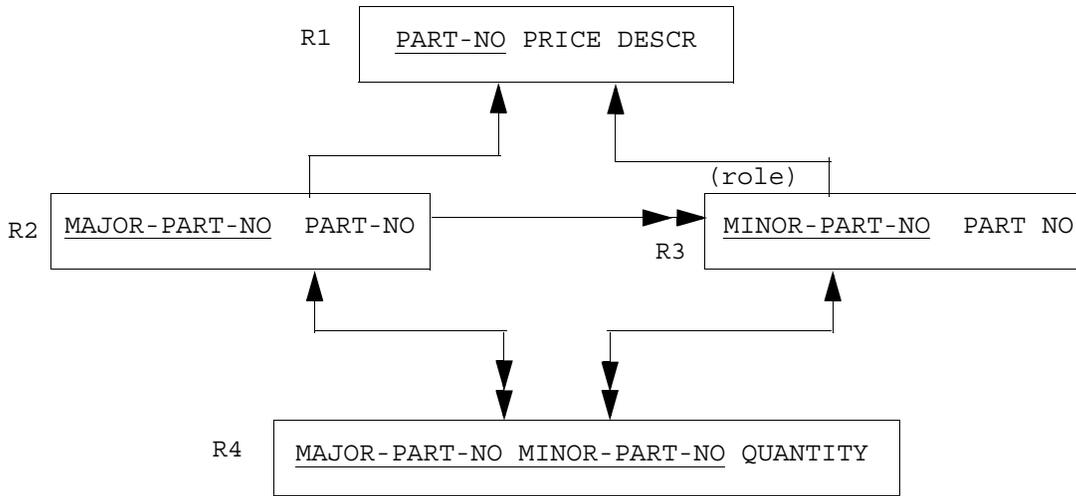
and these relations would be generated by the DesignManager software:

```

R1 (PART-NO, PRICE, DESCR)
R2 (MAJOR-PART-NO, PART-NO)      (role relation)
R3 (MINOR-PART-NO, PART-NO)      (role relation)
R4 (MAJOR-PART-NO, MINOR-PART-NO, QUANTITY)
    
```

No distinction is made in the userview between a pseudo-FD and an ordinary FD, and the DesignManager software handles them in exactly the same way. R2 and R3 are role relations which serve to identify a MAJOR-PART-NO or a MINOR-PART-NO as a PART-NO and thus provide access paths from R4 to R1.

As in the stock exchange example, the generated relations can be depicted graphically using arrows to represent pointers:



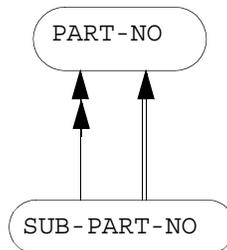
In this example, one might ask whether or not you would achieve the same result by defining only one subcategory, say SUBPART-NO (or MINOR-PART-NO) of PART-NO, and thus only one pseudo-FD, and specifying these dependencies instead:

```
PART-NO --->> SUBPART-NO
PART-NO+SUBPART-NO ---> QUANTITY
```

The answer is that it would have worked just as well (and would probably have been aesthetically more satisfying). Nevertheless, definition of the *associated* subcategory (MAJOR-PART-NO) is recommended as the safer course. Suppose, in the same example, there had been an added requirement to find all the parts which contain (as an immediate subpart) a given subpart. Defining only one subcategory, SUBPART-NO, for the data element PART-NO would result in an inconsistency in the workbench design area, namely:

```
SUBPART-NO ---> PART-NO      (the pseudo-FD)
SUBPART-NO --->> PART-NO    (the required MVD)
```

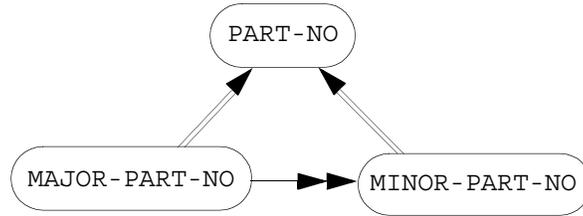
as shown in this diagram:



As a consequence, one or the other of the dependencies would be rejected by DesignManager during processing of the MERGE command. On the other hand, by introducing the two subcategories, MINOR-PART-NO and MAJOR-PART-NO, this MVD is specified:

```
MINOR-PART-NO --->> MAJOR-PART-NO
```

which provides the required access without causing an inconsistency:



In general, when handling problems similar to those described in the examples above, these guidelines can be useful for deciding how many subcategories (if any) should be defined:

- If an MVD is required to map the domain of a data element onto one of its subdomains or to map a subdomain onto the domain or onto another subdomain (for example, part to subpart, subpart to containing part, course to prerequisite course, or department manager to department member), the user should define two subcategories, one for the required subdomain and the other for the corresponding subdomain (major part, current course, department subordinate) which, by implication, is *associated* with the required subdomain. Instead of this specification:

```

PART NO ---> SUBPART-NO
SUBPART-NO ---> PART-NO
SUBPART-NO ==> PART-NO
  
```

(which, as indicated above, produces an inconsistency in the workbench design area), this is specified:

```

MAJOR-PART-NO ---> MINOR-PART-NO
MINOR-PART-NO ---> MAJOR-PART-NO
MAJOR-PART-NO ==> PART-NO
MINOR-PART-NO ==> PART-NO
  
```

Similarly:

```

CURRENT-COURSE-NO ---> PREREQ-COURSE-NO
CURRENT-COURSE-NO ==> COURSE-NO
PREREQ-COURSE-NO ==> COURSE-NO
  
```

is preferable to:

```

COURSE-NO ---> PREREQ-COURSE-NO
PREREQ-COURSE-NO ==> COURSE-NO
  
```

Again:

```

DEPT-MANAGER ---> DEPT-SUBORDINATE
DEPT-MANAGER ==> DEPT-MEMBER
DEPT-SUBORDINATE ==> DEPT-MEMBER
  
```

should be specified instead of:

```

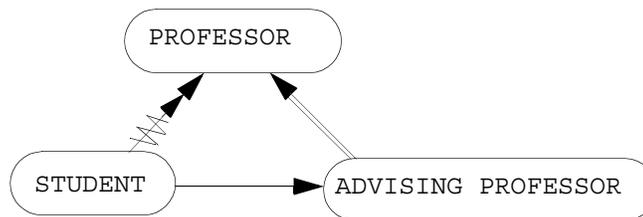
DEPT-MANAGER ---> DEPT-MEMBER
DEPT-MANAGER ==> DEPT-MEMBER
  
```

which, as in the first example, would result in an inconsistency. The first two examples (parts and courses) are BOMP problems, whereas the department member example is not (however, it can be converted to a BOMP problem by substituting EMPLOYEE for DEPT-MEMBER, MANAGER for DEPT-MANAGER, and SUBORDINATE for DEPT-SUBORDINATE, where MANAGER is used to describe someone who performs the generic function of managing rather than as a title; a subordinate of a manager could also be a manager with subordinates).

- Even if no dependency holds between the domain of a data element and one of its subdomains (other than a pseudo-FD from the subdomain to the domain) or between subdomains, it still may be necessary to define an added subcategory for the data element itself. For example, suppose that each student at a university takes courses from many professors but has only one advising professor. If only one subcategory is defined to portray the role of a professor as an advising professor:

```
STUDENT --->> PROFESSOR
STUDENT ---> ADVISING-PROFESSOR
ADVISING-PROFESSOR ==>> PROFESSOR
```

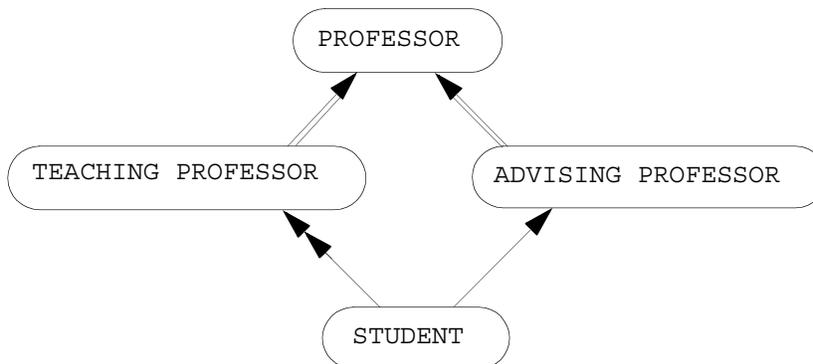
then DesignManager would reject the MVD during processing of the DESIGN command (see "[Step 2: Removal of Redundant Dependencies](#)" on page 140 of "[The DesignManager Design Procedure](#)" on page 138) as inconsistent with the two FDs, as shown in this diagram:



The solution is to define a second subcategory, say TEACHING-PROFESSOR, to represent the data element PROFESSOR itself in the MVD from STUDENT. The MVD becomes:

```
STUDENT --->> TEACHING-PROFESSOR
```

and the inconsistency is averted, as indicated by this diagram:



Recall that in the first example given, the stock exchange application, two subcategories of MEMBER-FIRM were defined to resolve the dilemma caused by the need for two FDs from BARGAIN-NO to MEMBER-FIRM. Even if the data element BARGAIN-NO had not been required for the application, both subcategories of MEMBER-FIRM would still have been defined. This is because a BOMP problem is implicit in the application; that is, an MVD from seller firms to buyer firms evidently must hold whether or not it is specified (as does the reverse MVD). So, by the first guideline given above, it is best to define both subcategories, SELLER-FIRM and BUYER-FIRM, rather than defining (say) just BUYER-FIRM and specifying the MVD:

MEMBER-FIRM --->> BUYER-FIRM

A few words of caution are in order at this point concerning the use of subcategories and pseudo-FDs in role situations. It should be observed as a hard and fast rule that:

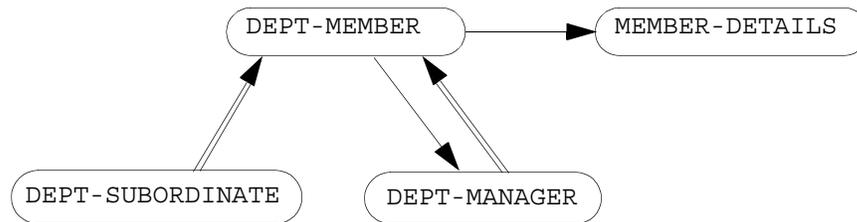
- An FD *never* be specified either from a data element to one of its subcategories or between two of its subcategories.
- An MVD *never* be specified from a subcategory of a data element to the data element itself.

To illustrate this rule, consider the department member example again, with these dependencies given:

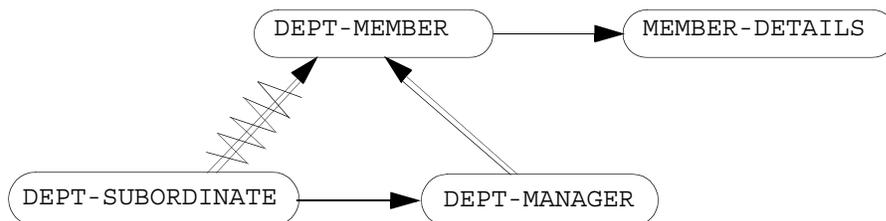
DEPT-MANAGER ===> DEPT-MEMBER
 DEPT-SUBORDINATE ===> DEPT-MEMBER
 DEPT-MEMBER ---> MEMBER-DETAILS

where MEMBER-DETAILS includes specifics about a department member such as home address, salary, and date of birth. The rule stated above implies that none of these dependencies should be specified with the given ones:

- DEPT-MEMBER ---> DEPT-MANAGER, because, in conjunction with the first pseudo-FD above, it would cause DesignManager to treat DEPT-MEMBER and DEPT-MANAGER as equivalent data elements, as indicated in this diagram:

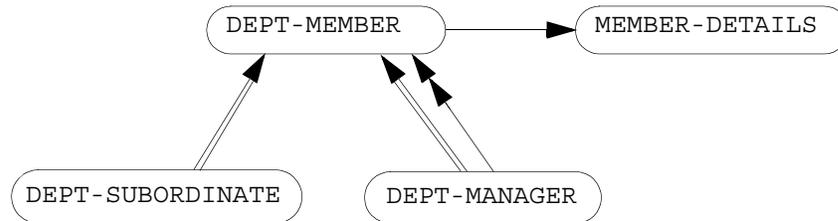


- DEPT-SUBORDINATE ---> DEPT-MANAGER, because, in conjunction with the first pseudo-FD, it would cause the software to eliminate the second pseudo-FD as transitively redundant, as indicated in this diagram:



As a consequence, access would be blocked from DEPT-SUBORDINATEs to the corresponding value of DEPT-MEMBERs and thus to their own MEMBER-DETAILS. Given DEPT-SUBORDINATEs, only the details pertaining to their DEPT-MANAGER would be obtainable, which is clearly unacceptable.

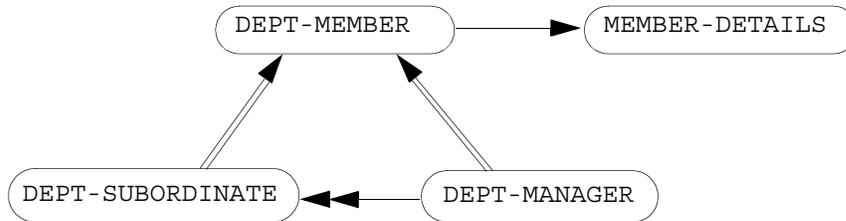
- DEPT-MANAGER --->> DEPT-MEMBER, because it conflicts with the first pseudo-FD given above, as shown in this diagram:



This would cause DesignManager, in processing the MERGE command, to eliminate one or the other of the dependencies in order to resolve the conflict and to flag the workbench design area as containing inconsistent userviews (see ["Merging Userviews into the Workbench Design Area" on page 130](#)). As indicated earlier, in the first guideline to determining the number of subcategories required for an application, the correct alternative is the MVD:

DEPT-MANAGER ---->> DEPT-SUBORDINATE

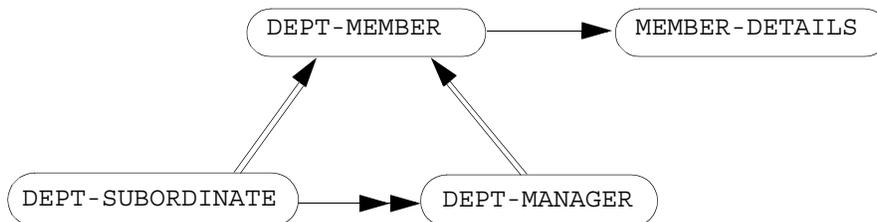
which leads to the this diagram:



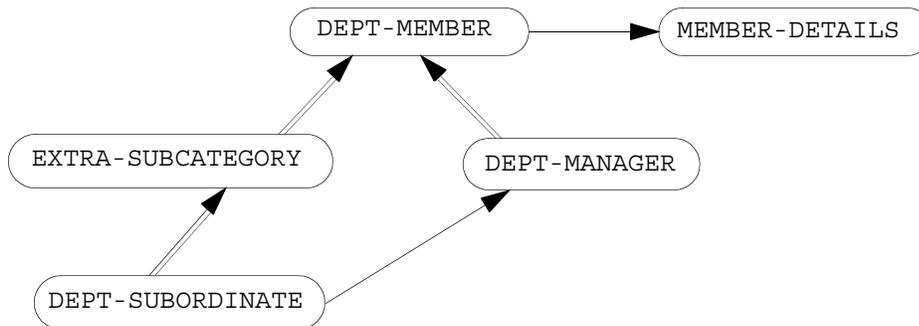
One might ask if there is a feasible alternative to either of these FDs, since they would cause the software to produce misleading or incorrect results:

DEPT-MEMBER ----> DEPT-MANAGER
 DEPT-SUBORDINATE ----> DEPT-MANAGER

One possible alternative would be an MVD from DEPT-SUBORDINATE to DEPT-MANAGER (whether or not the reverse MVD from DEPT-MANAGER to DEPT-SUBORDINATE is specified), as shown in this diagram:



Although the generated design would not indicate the desired FD, nevertheless, in any physical implementation, the indicated multivalued access path would in effect prove to be functional. Recall that an FD is a special case of an MVD (see "[Structure and Definitions](#)" on page 124). A second alternative would be insertion of an artificial subcategory between DEPT-SUBORDINATE and DEPT-MEMBER, as shown in this diagram:



Note: _____

The FD from DEPT-SUBORDINATE to DEPT-MANAGER is now permissible.

Expressed another way, the precautionary rule given above says that:

- FDs should not be specified between subcategories, only MVDs.
- No dependency other than a defining pseudo-FD should be specified between a data element and any of its subcategories.

7

Optional Additional Facilities

The basic DesignManager program provides all the facilities necessary for these functions:

- Creating, maintaining, and using the modeling dictionary
- Generating logical database design models from dictionary-held information
- Reporting on designs

As explained in [Chapter 1, "Introduction to DesignManager," on page 1](#), further modules can be incorporated into the basic program to provide additional facilities. Selection can be made from among these modules so that each installation of DesignManager can be provided with just those features of the product that are required by the organization in which it is installed.

The basic DesignManager program is called the DesignManager nucleus, and is identified by the code DSR-DS01. The modules constituting DesignManager optional additional facilities form a number of *selectable units*, each of which is similarly identified by a code, which is shown below in brackets after the selectable unit name.

This chapter summarizes each optional additional facility that is documented in its own individual facility publication. These optional facilities do not have individual publications:

- The TSO Interface (DSR-TP37) which is described in *ASG-Manager Products Installation in OS Environments*.
- The ICCF Interface (DSR-TP40) which is described in *ASG-Manager Products Installation in DOS Environments*.
- The CMS interface (DSR-TP38) which is described in *ASG-Manager Products Installation in OS Environments*.
- The Interactive Front-end facility (DSR-FE30) which is described in [Chapter 9, "Interactive Front-end Facility," on page 179](#).
- The User Printer Graphics facility (DSR-UD31) which is described in [Chapter 10, "User Printer Graphics \(DSR-UD31\)," on page 209](#).
- The DesignManager Manager Product Integrator facility (DSR-MP03). The use of this facility for DesignManager/ASG-DataManager integration is described in [Chapter 8, "DesignManager/DataManager Integration," on page 169](#).

For the availability of the various facilities and their constituent selectable units, reference should be made to the most recent issue of the DesignManager evaluation document entitled *How to Evaluate, Configure and Cost*.

User Formatted Output (DSR-UD30)

The User Formatted Output facility enables users to control the format and content of the DesignManager design mode reports, and extends the information that can be output.

The facility provides an additional modeling dictionary member type, FORMAT. FORMAT members are used to store report format specifications. They include facilities for specifying the layout of the report, the content of the report, the title of the report, the inclusion of selected information from the modeling dictionary, and the positioning of the fixed text and variable parameters.

The User Formatted Output facility extends the options of the format-selection clause of certain design mode commands so that a FORMAT member which describes the format of a particular report can be specified in the command. These commands are DESIGN, REPORT, and, if the User Printer Graphics facility is installed, PLOT.

The facility also provides a preferred set of standard FORMAT members, including one for each report category, in detail and in summary. Users may copy the supplied FORMAT members to new named members (using the dictionary mode facilities) and edit them to reflect their own report requirements. Alternatively, users may define their own FORMAT members to describe the reports they require. Over a period of time, a library of FORMAT members can be built up in the modeling dictionary.

The User Formatted Output facility is a design mode facility, and produces reports on data in the workbench design area. Information held in the modeling dictionary that is required in the reports is accessed automatically.

When the User Formatted Output facility is installed, the keyword FORMATS may be used in all DesignManager commands where the variable *optional-additional-member-type* appears.

Enterprise Modeling (DSR-EM10)

The Enterprise Modeling facility is intended for DesignManager users who employ a top-down approach to database design, either instead of or in conjunction with a bottom-up approach (exemplified by the procedures described in [Chapter 6, "DesignManager Procedures and Use," on page 123](#)).

The basis for DesignManager enterprise modeling is the entity, introduced both as an additional modeling dictionary member type and as an additional data type in the workbench design area. The user can specify any of these as options in an ENTITY member:

- An identifier of the entity, as a set of one or more data elements
- Attributes of the entity, composed of functionally and/or multiply determined sets of data elements
- Associations from the entity to other entities, reflecting implied dependencies between entity identifiers, whether or not identifiers have been specified
- Subentities of the entity

For top-down modeling, the entities of an enterprise, along with some or all of its associations to other entities of the enterprise, can be defined and entered in the modeling dictionary before any attributes are specified. The identifier and attributes can be added gradually as the details of the enterprise become more clearly defined. The input for a top-down approach, both entities and associations, often is supplied by middle and top-level management, who are frequently in the best position to grasp the overall picture.

Entities can be merged into the workbench design area (in the form of functional and multivalued dependencies) and a DESIGN command used to produce a third normal form logical schema. This can be done even if no identifier and attributes are specified for some or all of the entities; for each such entity, DesignManager automatically creates and merges a functional dependency in the workbench design area using default data elements generated from the entity name.

The Entity Report provides output describing the entities in the workbench design area.

DesignManager installation publications describe the Installation of the Enterprise Modeling facility.

ASG-DesignManager Enterprise Modeling documents the facility.

Load Factor Calculation (DSR-PH10)

The Load Factor Calculation facility provides the DesignManager user with the means of modeling the physical access times of data elements of the logical database model with respect to the userviews in which those elements appear. The modeling process is carried on in parallel with the main database design process.

The facility provides a command, CALCULATE, to carry out load factor calculations; load factors are always calculated for all data elements referenced by userviews that are present in the workbench design area when the calculations are carried out. The user can then use the REPORT command to output all, or a selection of, load factor values in a report format (*Load Factor Analysis Report*). If the user's DesignManager installation also includes the User Formatted Output facility (DSR-UD30), then the user may also specify their own format for output of load factors (this facility is described in *ASG-DesignManager User Formatted Output*).

8

DesignManager/DataManager Integration

The bulk of the DesignManager documentation describes the features, capabilities, and facilities available when DesignManager is installed in a non-integrated (stand-alone) environment. This chapter documents the effects on the basic and optional additional facilities of both products when DesignManager is integrated with ASG-DataManager (herein called DataManager) by way of two Manager Product integrator facilities (Selectable Units DSR-MP03 and DMR-MP01 respectively). Consider these effects in seven groups:

- Effects on the DesignManager dictionary mode commands
- Effects on the DesignManager design mode commands
- Effects on the DataManager commands
- Effects on DesignManager optional additional facilities
- Effects on DataManager optional additional facilities
- Effects on language and coding

["Effects on DesignManager Dictionary Mode Commands" on page 170](#) through ["Effects on Language and Coding" on page 178](#) describe these groups of effects. Throughout the descriptions, a knowledge of DataManager is assumed.

The effects of integration depend on which of the two products the user signs onto at the beginning, and therefore for the duration, of a run. If the user signs on to DesignManager, all the facilities and effects described in this publication are available, provided that the relevant selectable units are installed. If the user signs on to DataManager, all the DataManager facilities are available (and therefore DesignManager dictionary mode facilities and member types; see ["Effects on DesignManager Dictionary Mode Commands" on page 170](#)), but no DesignManager design mode facilities (including optional additional facilities available in design mode) are available.

[Chapter 6, "DesignManager Procedures and Use," on page 123](#) describes some aspects of the joint use of DesignManager and DataManager in integrated environment.

In these sections, all references to ENTITY members and member types are applicable only for users who have the optional Enterprise Modeling facility (Selectable Unit DSR-EMI0) installed. The keyword DATA-ELEMENTS refers generically to ITEM and/or GROUP member types; similarly DATA-VIEWS refers to USERVIEW and/or ENTITY member types.

Effects on DesignManager Dictionary Mode Commands

The integration of DesignManager and DataManager is primarily through the dictionary and dictionary management software. The effect upgrades the capabilities available in DesignManager's dictionary mode to the full capabilities of DataManager.

In integrated DesignManager/DataManager environments, users should ignore all dictionary mode commands documented for stand-alone DesignManager installations and use their existing DataManager documentation in light of the effects and additional capabilities described in ["Effects on DataManager Commands" on page 172](#).

Effects on DesignManager Design Mode Commands

The effects of integration on the DesignManager design mode commands are limited to the DesignManager MERGE command (see ["The DesignManager MERGE Command" on page 170](#)). Certain DataManager commands are also available in design mode (see ["DataManager Commands Available In Design Mode" on page 171](#)).

The DesignManager MERGE Command

The basic MERGE command available to stand-alone users and documented in ["Design Mode Command Specifications" on page 73](#), is extended in integrated DesignManager/DataManager installations to take advantage of the additional dictionary mode capabilities DataManager provides. The extended command is shown below.

Note: _____

It is important to note that the MERGE command must not be used if the DataManager name concatenation capability is SWITCHED ON.

MERGE Format

```
MERGE [NO-VERIFY] { MEMBERS member-name [, member-name] ... }
                   { KEPT-DATA
                   { i }
                   { . }
```

MERGE Remarks

With a single status dictionary:

1. The remarks given for the MERGE command in ["Design Mode Command Specifications" on page 73](#), apply when using the MERGE [NO-VERIFY] MEMBERS member-name, member-name,... form of the command.
2. When specifying KEPT-DATA, the action of the command is as if those member names accumulated in a list in main storage as a result of a previous dictionary mode KEEP command (or KEEP command and one or more ALSO KEEP commands) entering sequentially in a MEMBERS clause.

If the Audit and Security facility is installed:

3. Acceptance or rejection of the MERGE command is in respect of each individual member named in the MEMBERS clause or whose name is held in the KEPT-DATA list, and/or of each member directly contained in a member so named or held. The command accepts for each such member to which the user has access, and rejects with an appropriate warning for each member to which access is not permitted.

With a multi-status dictionary:

4. If a member named in the MEMBERS clause, or whose name is held in a KEPT-DATA list, is a verified USERVIEW, ENTITY, or VIEWSET in the current status, the MERGE command operates within the current status as defined in [remark 1 on page 170](#) through [remark 3](#).
5. If a member named in the MEMBERS clause, or whose name is held in a KEPT-DATA list, is not a verified USERVIEW, ENTITY, or VIEWSET in the current status, but appears as such a member in one or more earlier frozen statuses, the MERGE command operates as defined in [remark 1 on page 170](#) through [remark 3](#) in the most recently frozen status of the earlier frozen statuses in which the member so appears.
6. If a member named in the MEMBERS clause, or whose name is held in a KEPT-DATA list, is not a verified USERVIEW, ENTITY, or VIEWSET in the current status or in any earlier frozen status, the MERGE command operates as defined in the MERGE command specifications in ["Design Mode Command Specifications" on page 73](#).

DataManager Commands Available In Design Mode

Four DataManager commands are available in design mode:

- PERFORM
- SKIP
- SPACE
- SWITCH

The SKIP and SPACE commands are available as documented in the *ASG-Manager Products Dictionary/Repository User's Guide*. PERFORM and SWITCH are available as documented in the *ASG-Manager Products Dictionary/Repository User's Guide* but with the restrictions stated here.

The PERFORM COMMAND-STREAM member-name form of the DataManager PERFORM command may be issued in design mode (and in dictionary mode as for all DataManager commands; see ["Effects on DataManager Commands" on page 172](#)) but no selection of the members on which the command operates is available within the command. Here is the syntax of the PERFORM command in design mode:

```
PERFORM COMMAND-STREAM member-name { ; }
                                     { . }
```

The PERFORM input-line, input-line,... form of the command is not accepted in design mode.

COMMAND-STREAM member types PERFORMed in design mode can include design mode, mode and dictionary mode commands but cannot include PERFORM commands (see ["Effects on Dictionary Member Types" on page 175](#)).

In design mode, the DataManager SWITCH command is only available for switching design mode messages on and off and switching output to a different output file; the ON NAME-CONCATENTATION clause is not available. Here is the syntax for the command in design mode:

```
SWITCH { [ { OFF } MESSAGES [ ALSO ] [ [ LEVELS ] I [ , W ] ] } ; {  
         { ON }           [ W [ , I ] ] } . {  
         [ NUMBERS m [ TO n ] [ , m [ TO n ] ] ... ] ... }  
         OUTPUT TO ddname [ AND ddname ] }
```

where:

m is an unsigned integer in the range 50,000 to 54,999.

n is an unsigned integer greater than the last preceding *m* and not greater than 54,999.

A full specification of the SWITCH command is given in *ASG-Manager Products Dictionary/Repository User's Guide*.

Note: _____

Users should note that the setting up of the local file name *ddname* in the OS job control statements differs from that used when *ddname* is set up under DataManager.

The command SWITCH OUTPUT TO DMOUT returns output to the main device.

Effects on DataManager Commands

In integrated DesignManager/DataManager environments, the full range of DataManager commands (with the exception of the SWITCH command) replaces the DesignManager dictionary mode commands available to stand-alone installations. Users should therefore refer to existing DataManager documentation for all dictionary mode commands, elements of which are extended as shown in the table at the end of this section.

The SWITCH command operates as documented in *ASG-Manager Products Dictionary/Repository User's Guide* with the exception that ddname must be set up according to the DesignManager installation procedure.

Note: _____

The permissible message numbers that can be switched on and off is unchanged; that is, only message numbers in the range 1 to 32,767 can be used (see also ["DataManager Commands Available In Design Mode" on page 171](#)).

Extensions to the DataManager Commands in Integrated DesignManager/DataManager Installations

| Command | Element Affected | Additional Keywords | |
|----------------|---------------------------------|---------------------------------|------------------------|
| BULK | member-type | <u>DATA-ELEMENTS</u> | |
| | | <u>DATA-VIEWS</u> | |
| | | <u>USERVIEWS</u> | |
| | | <u>ENTITIES</u> | |
| | | <u>VIEWSETS</u> | |
| | optional-additional-member-type | | |
| GLOSSARY | category | <u>DATA-ELEMENTS</u> | |
| | | <u>DATA-VIEWS</u> | |
| | | <u>USERVIEWS</u> | |
| | | <u>ENTITIES</u> | |
| | | <u>VIEWSETS</u> | |
| | | optional-additional-member-type | |
| | | extract | <u>LEFT-HAND-SIDE</u> |
| | | | <u>LHS</u> |
| | | | <u>RIGHT-HAND-SIDE</u> |
| | | | <u>RHS</u> |
| GLOSSARY | member-type | <u>DATA-ELEMENTS</u> | |
| | | <u>DATA-VIEWS</u> | |
| | | <u>USERVIEWS</u> | |
| | | <u>ENTITIES</u> | |
| | | <u>VIEWSETS</u> | |
| | optional-additional-member-type | | |

Extensions to the DataManager Commands in Integrated DesignManager/DataManager Installations

| Command | Element Affected | Additional Keywords |
|----------------------|-------------------------|--|
| LIST | index-name-type | <u>DATA-ELEMENTS</u> <u>DATA-VIEWS</u> <u>USERVIEWS</u> <u>ENTITIES</u> <u>VIEWSETS</u> optional-additional-member-type |
| PERFORM | index-name-type | <u>DATA-ELEMENTS</u> <u>DATA-VIEWS</u> <u>USERVIEWS</u> <u>ENTITIES</u> <u>VIEWSETS</u> optional-additional-member-type |
| REPORT | member-type | <u>DATA-ELEMENTS</u> <u>DATA-VIEWS</u> <u>USERVIEWS</u> <u>ENTITIES</u> <u>VIEWSETS</u> optional-additional-member-type |
| WHICH | category | <u>DATA-ELEMENTS</u> <u>DATA-VIEWS</u> <u>USERVIEWS</u> <u>ENTITIES</u> <u>VIEWSETS</u> optional-additional-member-type |
| WHICH and WHAT | VIA clause | <u>LEFT-HAND-SIDE</u> <u>LHS</u> <u>RIGHT-HAND-SIDE</u> <u>RHS</u> |

Notes

1. *Optional-additional-member-type* refers to additional member types provided in DesignManager optional additional facilities.
2. DATA-ELEMENTS refers to ITEM and GROUP members.
3. DATA-VIEWS refers to USERVIEW and ENTITY members.

Effects on Dictionary Member Types

The ITEM and GROUP member types are common to DesignManager and DataManager. In DesignManager, the definition syntax for these member types provides all the functionality required for defining the data elements that will be part of the database for which a logical database design model is to be generated. In DataManager, the definition syntax is greatly extended to meet the definitional requirements of all aspects of data administration, including the generation of data descriptions for application programs operating on a database. In integrated environments, the full definition capabilities of DataManager are available.

The DataManager capability to define alias types within the ALIAS clause may be applied to all DesignManager member types, including those provided in DesignManager optional additional facilities.

In addition to containing streams of dictionary mode commands, the COMMAND-STREAM member type may also include DesignManager design mode and mode commands, although such members can only be executed by a PERFORM command issued in design mode.

Note: _____

Any command or data definition stored in a COMMAND-STREAM member must start on a new input line.

DesignManager USERVIEW, ENTITY, and VIEWSET members can be validly referred to in the CONTAINS, INPUTS, OUTPUTS, UPDATES, PASSING and PARAMETERS clauses of the DataManager process member types.

Effects on DesignManager Optional Additional Facilities

The effects of integration on DesignManager optional additional facilities are concerned with the member types provided and the interaction of the provided commands with the dictionary, as described in ["User Formatted Output \(DSR-UD30\)" on page 176](#) and ["User Printer Graphics \(DSR-UD31\)" on page 176](#).

User Formatted Output (DSR-UD30)

As with other DesignManager member types, the data definition syntax of the FORMAT member type is extended so that alias types may be defined in the ALIAS clause.

When a FORMAT member is supplied in the format-selection clause of a design mode DESIGN or REPORT command, then:

- With a single status dictionary, the commands operate as defined in *ASG-DesignManager User Formatted Output*.
- With a multi-status dictionary, if format-name has no encoded record in the current status, the encoded record of format-name in the most recently frozen of the earlier frozen statuses in which it has a record is used.
- If the DataManager Audit and Security facility (DMR-DD3) is installed:
 - Use of format-name is subject to access security levels.
 - Access to information held in the dictionary and specified in an extract clause in the FORMAT member is subject to access security levels in respect of each member so accessed.

User Printer Graphics (DSR-UD31)

Integration has no effect on the User Printer Graphics facility unless the User Formatted Output facility is also installed. In that case, specification of format-name in the format-selection clause of the design mode PLOT command is affected as described for the design mode DESIGN and REPORT commands in ["User Formatted Output \(DSR-UD30\)" on page 176](#), that is:

- With a single status dictionary, the commands operate as defined in *ASG-DesignManager User Formatted Output*.
- With a multi-status dictionary, if format-name has no encoded record in the current status, the encoded record of format-name in the most recently frozen of the earlier frozen statuses in which it has a record is used.
- If the DataManager Audit and Security facility (DMR-DD3) is installed:
 - Use of format-name is subject to access security levels.
 - Access to information held in the dictionary and specified in an extract clause in the FORMAT member is subject to access security levels in respect of each member so accessed.

Effects on DataManager Optional Additional Facilities

This section describes the process member types and user defined syntax (UDS) for basic and optional additional facilities used in integrated environments.

Process Member Types

DesignManager USERVIEW, ENTITY, and VIEWSET members may be referenced in the CONTAINS, INPUTS, OUTPUTS, UPDATES, PASSING, and PARAMETERS clauses of any process member type provided by a DataManager optional additional facility.

User Defined Syntax (UDS)

Existing User Defined Syntax (herein called UDS) users will need to install new UDS specifications to include the new DesignManager member type keywords.

DesignManager USERVIEW, ENTITY, and VIEWSET members may be referenced in the CONTAINS, INPUTS, OUTPUTS, UPDATES, PASSING, and PARAMETERS clauses of any user defined process member type, unless these clauses are excluded by the UDS specification applied to the dictionary.

Any user defined member types that are based on the DataManager BASIC member types GROUP and ITEM, may be used in the dependencies-clauses of a DesignManager USERVIEW member and in the IDENTIFIER, ONE-ATTRIBUTES, or MULTI-ATTRIBUTES clauses of a DesignManager ENTITY member.

When UDS is installed, UDS support is provided for DesignManager member types, in addition to those of DataManager as indicated here:

- User defined member types may be based on ITEM and GROUP.
- User defined member types may be based on USERVIEW, VIEWSET and, if the Enterprise Modeling facility (selectable unit DSR-EM10) is installed, ENTITY. The syntax of the user defined member type must be the same as the member type on which it is based. The user defined member type, when brought onto the workbench (using the MERGE command), is identified and processed in the same manner as the member type on which it is based.

In integrated environments with user defined member types that conflict with the DesignManager BASIC and ADDITIONAL member types, users must decide whether to:

- Alter the conflicting user defined member type keyword so that there is no conflict with the DesignManager BASIC and ADDITIONAL member type keywords. (This may be done by installing a new UDS table specification with a new keyword replacing the conflicting keyword, and PERFORMing a series of MODIFY commands to change the conflicting keyword to the replacement keyword in the appropriate members.) Or:
- Maintain two separate dictionaries:
 - A data dictionary with the UDS specification containing (among others) the conflicting user defined member type keywords and none of the DesignManager BASIC and ADDITIONAL member type keywords.
 - A modeling dictionary with a UDS specification that includes the DesignManager BASIC and any ADDITIONAL member types and has no conflicting member type keywords.

In situations where a DesignManager member type keyword has been specified for a user defined member type that serves the same purpose as the DesignManager member type, users can install a UDS specification that substitutes the DesignManager member type for the previously user defined member type and MODIFY the appropriate members to conform to the DesignManager syntax for these member types.

TERMINOLOGY has been proposed as a BASIC or optional additional member type for future releases of DesignManager. It is mentioned here to avoid conflict between future DesignManager BASIC and ADDITIONAL member types and user defined member types. While every effort is made to limit the introduction of new DesignManager member types, no guarantee is given that this member type will be used.

Effects on Language and Coding

The rules for coding and input of commands and data definition statements follow the rules given in [Chapter 2, "DesignManager Language and Coding," on page 15](#) when the user signs on to DesignManager at the beginning of a run, and follows the rules given in *ASG-DataManager User's Guide* when the user signs on to DataManager.

9

Interactive Front-end Facility

Overview of the Interactive Front-end

The interactive Front-end optional additional facility (selectable unit DSR-FE30) allows you to examine interactively output from DesignManager commands. You can copy selected parts of the output to a separate (save) area, which you can then use as:

- A quick reference during your interactive session
- Input for DesignManager commands

There are 30 commands available for use exclusively with the Interactive Front-end. Using these commands you can:

- Manipulate the screen display so that you can examine a selected screen either in the output area or save area (see ["Using the ESCAPE Character" on page 180](#) and ["Commands to Examine the Display Areas" on page 183](#)).
- Edit the information in the save area, so that it can be used as input to DesignManager, or simply for reference (see ["Commands to Edit the Save Area" on page 184](#)).
- Call for a standard stream of Front-end commands, which may include prompting messages and pauses that enable you to input information (see ["Macro Usage Commands" on page 186](#)).
- Submit the contents of the save area as commands to DesignManager for execution (see ["The EXECUTE Command" on page 182](#)).
- Exit from the interactive session at any point (see ["Using the ESCAPE Character" on page 180](#)).

The output from all DesignManager design mode commands (see ["Design Mode Command Specifications" on page 73](#)) is available for interactive examination by the Front-end: output from all DesignManager dictionary mode commands (see ["Dictionary Mode Command Specifications" on page 28](#)), except the MODIFY command, is similarly available. The edited screen-displayed output can then be used as input for subsequent DesignManager commands. To cater for the examination and editing capabilities, there are two separately displayed areas: the output area and the save area, respectively. The user may switch the display between the two areas.

For the purposes of interactive examination, the output from a DesignManager command is broken up into sequential buffers, as many as are necessary to contain the entire output. These buffers may only be examined one at a time. It is recommended that the buffer sizes be maximized for ease of viewing and editing. Tailor the size of the output buffers using the macro LBUF 1 (see *ASG-Manager Products Installation in OS Environments* or the *ASG-Manager Products Installation in DOS Environments*).

By entering a DesignManager command, the first buffer of output goes into the output area, and the first screen of the output area displays, showing the first lines of the DesignManager command output. Then use Interactive Front-end commands to copy parts of the output area to the save area, to edit the save area, and to manipulate the display of various parts of the output or save areas, as their contents are examined.

If another buffer of output is to follow, the last screen of the buffer currently in the output area has this message as its final line:

```
*** MORE OUTPUT FOLLOWS . KEY & TO CONTINUE ***
```

The ampersand (&) character (see ["Command Specifications" on page 187](#)) in this message is the default setting for the ESCAPE character. This setting may be changed, for the duration of an Interactive Front-end run, using the SET command (see ["Command Specifications" on page 187](#)). The default of the ESCAPE character may be altered permanently by tailoring the LOPT1 macro (see the *ASG-Manager Products Installation in OS Environments* or the *ASG-Manager Products Installation in DOS Environments*). If the default character is tailored then the new default is output in the above message.

By entering the ESCAPE character, the next buffer of output from the DesignManager command goes into the output area, and the previous buffer is no longer available. However the contents of the save area are retained; by editing in the save area, unwanted data is removed and the data required from each buffer of output is retained for later use. The user may continue with interactive examination and editing of successive buffers of output as each moves into the output area. Finally the last buffer of the entire output is brought into the output area. When it displays, the last screen of this buffer shows a prompt for the next DesignManager command. Entering this next DesignManager command (whether it is valid or not) clears the output area.

When installing the Interactive Front-end facility, and output from a DesignManager command displays on the screen, DesignManager accepts only the Interactive Front-end commands, unless this prompt appears. This ensures that all buffers of output are processed, and that any messages associated with and appearing at the end of the output are seen by the user. However, this safety feature may be overridden by prefixing the DesignManager command by the ESCAPE character. ["The Display Areas" on page 181](#) discusses the two display areas in more detail.

Using the ESCAPE Character

The ESCAPE character serves these two purposes:

- To force the execution of a DesignManager command when the output displayed is not the last screen of the last buffer, but the last buffer has been brought into the output area.
- To cause the next buffer of output to be brought into the output area.

When the last buffer is in the output area and the ESCAPE character is followed, with or without intervening spaces, by a valid DesignManager command, that command is executed irrespective of whether the displayed screen is the last screen of the buffer (where the prompt for the next DesignManager command is shown), or whether the current display is from the output or save area. Entering this ESCAPE character, DesignManager command sequence, causes the output area to clear; this occurs even if the DesignManager command is invalid. If it is not the last buffer of output in the output area, the DesignManager command is ignored, but the next buffer of output moves into the output area.

Whichever area currently displays, entering the ESCAPE character alone, when the last buffer has yet to move into the output area, causes the next buffer to move into the output area, and its first screen to display. This means that the prompt for entering the ESCAPE character (which appears on the last screen of the current buffer in the output area) need not appear in order that the next buffer of output be brought into the output area. If the output area contains the last buffer of output, then the output area clears and the prompt for the next DesignManager command displays.

The Display Areas

The two display areas are the output area and the save area. As stated in ["Overview of the Interactive Front-end" on page 179](#), DesignManager command output, which is to be examined and perhaps edited using the Interactive Front-end facility, breaks into sequential buffers. Successive buffers move into the output area. From here, parts of the output may be moved to the save area, to edit or save, as required. Both areas are examined via the screen display.

Unless tailored by the LBUF1 macro (see *ASG-Manager Products Installation in OS Environments* or *ASG-Manager Products Installation in DOS Environments*), the manual is full when either it contains 350 lines or 14,000 characters (whichever occurs first). The size of the output area is determined by the same parameter of this macro, and is therefore the same size as a buffer. Tailoring a different parameter of the LBUF1 macro (again, see the appropriate installation publication) determines the size of the save area.

All editing of the DesignManager command output is carried out in the save area; the current buffer of output in the output area is for examination only. When the first buffer is brought into the output area, the save area is filled with blanks. Copy output from the output area to the save area, where it may be edited. When a new buffer is brought into the output area, the contents of the save area remain unchanged. Using the DesignManager design mode command EXECUTE (see ["Command Specifications" on page 187](#)), the edited version of the output, available in the save area, can be used as input for subsequent DesignManager commands in the same interactive session.

The output area and save area screen display layouts differ only in one line: when the output area displays, the top line of the screen is blank; when the save area displays, the title *Save area* appears on this line.

If there are screens of output from the display area, which precede and/or succeed the currently displayed screen, then this is indicated in the top left-hand corner and/or bottom right-hand corner of the screen.

Error messages resulting from any invalid Interactive Front-end command operations appear, delimited by asterisks (***), on the top line of the screen.

Note: _____

If the save area currently displays, the error message replaces the *Save area* title for that particular screen-display.

Throughout this chapter the first line of output on the screen, displayed from either the output or save area, is known as the *current line*. After the execution of an Interactive Front-end command completes, the current line is re-determined as the first line of the new screen display. When the display is switched from one area to the other, the line that was current before the switch remains the current line when returning to the area.

The EXECUTE Command

Unlike the other commands available with the Interactive Front-end, the EXECUTE command is not used for interactive examination or editing of screen displayed output. It is a design mode command that is used to submit the contents of the save area as commands to DesignManager for execution.

Before issuing an EXECUTE command the save area must be edited so that it is in the form of a DesignManager command, or a series of DesignManager commands.

The uses of this capability obviously depends on the user, but here are examples of its usage:

- To add new members to the dictionary using ADD commands generated in design mode Logical Schema Reports.
- To set up a series of NAME RELATION commands from the output of a design mode LIST RELATIONS command.

Overview of Interactive Front-end Commands

Consider the commands provided by the Interactive Front-end facility for the examination and editing of output from DesignManager commands in these four groups:

- The :SET command
- Commands to examine the display areas
- Commands to edit the save area
- Macro usage commands

An overview of these four groups of commands is given in ["The :SET Command" on page 183](#) through ["Macro Usage Commands" on page 186](#). In ["Command Specifications" on page 187](#), their specifications are given in alphabetical sequence of command identifier: the notation used in format specifications is defined in ["Notation For Statement Formats" on page ix](#). ["Error Messages" on page 186](#) discusses error messages which can result from Interactive Front-end commands.

All Interactive Front-end commands have a PREFIX character as the first character of their identifier. The default setting for the PREFIX character as supplied is a colon (:). The PREFIX character may be changed, for the duration of an interactive session, by using the Interactive Front-end :SET command (see ["Command Specifications" on page 187](#)). The default may be altered permanently by tailoring the LOPT1 macro (see *ASG-Manager Products Installation in OS Environments* or the *ASG-Manager Products Installation in DOS Environments*). Command identifiers specified in this documentation use the default PREFIX character setting.

Depending on the environment, program function keys may be assigned to certain of the commands provided in the Interactive Front-end facility (see *ASG-Manager Products Installation in OS Environments* or *ASG-Manager Products Installation in DOS Environments*).

The :SET Command

The :SET command is used to alter, for the duration of an interactive session, the value of the ESCAPE character (standard default the ampersand (&) character, see also ["Using the ESCAPE Character" on page 180](#)) and/or the PREFIX character (standard default the colon (:) character, see also ["Overview of Interactive Front-end Commands" on page 182](#)).

Commands to Examine the Display Areas

The DesignManager Interactive Front-end facility provides a variety of commands for manipulating screen-displayed output for ease of examination. These are the commands:

- :BOTTOM
- :DOWN
- :LOCATE
- :NUMBERS
- :POSITION
- :SWITCH
- :TOP
- :UP
- :+
- :-

A detailed specification of each of these commands is given in ["Command Specifications" on page 187](#), however this section describes their general usage.

:BOTTOM command causes the last screen of the current display area to display. Thus allowing the user rapid access to information held on the final screen by simply issuing a single command.

:DOWN command causes the following screen of the current display area to display. An alternative form of the command :DOWN n causes the n+ 1th following screen to display. This allows the user to scroll downwards through the current display area as required.

:LOCATE command locates the first occurrence of a specified character string in the current display area. This command enables the user to search for and locate a particular character string either anywhere within the current display area, or optionally between particular columns within the display area. This latter option is particularly useful when tabular output displays.

:NUMBERS command allows the display of line numbers to be turned on or off. If line numbers display and the user's screen has a width of 80 characters, then the final four characters on each line are not visible on the screen. However, the command `:NUMBERS OFF` causes line numbers to be removed and all 80 characters on each line become visible. It is recommended that line numbers be switched off when viewing the display area and be switched on when knowledge of line numbers is required (see `:POSITION` command below).

:POSITION command causes the current display area to move so that a specified line becomes the top line on the screen. The command `:POSITION 10` causes line number 10 to become the first line on the screen. Obtain line numbers by using the `:NUMBERS ON` command described above.

:SWITCH command causes the screen display to switch from the current area to the other area (for example from the output area to the save area).

:TOP command causes the first screen of the current display area to display. This allows a user to return rapidly to the top of the display area simply by issuing a single command.

:UP command causes the preceding screen of the current display area to display. An alternative form of the command `:UP n` causes the $n+1$ th preceding screen to display. This allows the user to scroll upwards through the current display area as required.

:+n command and the **:-n command** cause the display of a screen n lines after and n lines before the current line respectively. When used in conjunction with the `:DOWN` and `:UP` commands any selected screen within the entire current display area can display.

Commands to Edit the Save Area

["The Display Areas" on page 181](#) describes the uses of the save area. These commands are available for editing the save area to prepare it for use:

- `:COPY`
- `:INSERT`
- `:ADD`
- `:DUPLICATE`
- `:CHANGE`
- `:SCHCHANGE`
- `:VERIFY`
- `:SHIFT`
- `:DELETE`
- `:WIPE`

:COPY command is used in conjunction with the **:INSERT** command to copy lines from either display area and to insert them into the save area as required. A specified range of lines is copied by issuing a **:COPY** command. These lines are then input into the save area in a specified position when the **:INSERT** command is issued. The copied lines can be inserted any number of times without issuing an additional **:COPY** command.

:ADD command is used to add lines to the save area in a specified position. The user simply specifies the position and the number of lines to add and then keys in and enters each individual line.

:DUPLICATE command is used to copy a specified number of times a single line that is already held in the save area. This is particularly useful when extending tables and fixed format diagrams.

:CHANGE command and the **:SCHANGE** command enable a particular character string to be replaced by another character string. Both commands provide the capability of:

- Changing all occurrences of the character string
- Changing the first occurrence of the character string
- Changing the character string on particular lines only
- Changing occurrences of a particular character string between specified columns only

This is the difference between the **:CHANGE** command and the **:SCHANGE** command:

- On completion of the **:CHANGE** command the position of the current line of the save area is unaltered
- For the **:SCHANGE** command the current line becomes the line after the last changed line

:VERIFY command is designed for use with the **:CHANGE** and **:SCHANGE** commands. If the command **:VERIFY ON** is entered then any lines altered due to subsequent **:CHANGE** and **:SCHANGE** commands display. Switch off this capability by entering the command **:VERIFY OFF**.

:SHIFT command is used to move lines to the left or to the right in the save area. The range of lines to move and the number of character places by which the lines are to move can be specified in the command. One of the features of this command is that any characters moved off the screen are deleted; thus any unwanted data (such as line numbers) at the beginning or the end of lines is permanently removed.

:DELETE command is used to delete complete lines from the save area. This command provides the capability of removing a specified range of lines from the save area.

:WIPE command is used to wipe the save area completely clear of data, so that it is available for new usage.

Macro Usage Commands

The macro usage commands available with the DesignManager Interactive Front-end facility are designed for use in a COMMAND-STREAM of Front-end commands. The macro usage commands are accepted when input individually, but their major usage is expected to be in COMMAND-STREAMs. These are the macro usage commands:

- :MACRO
- :FAIL
- :SKIP
- :PROMPT
- :READ
- :WRITE
- :MEND

:MACRO command is used to initiate processing of a named COMMAND-STREAM member that is held in the dictionary. The COMMAND-STREAM member that is called must contain Front-end commands only (excluding the EXECUTE and ampersand (&) commands).

:FAIL command causes a command specified within it to execute only if the command immediately before the :FAIL command is erroneous.

:SKIP command causes a specified number of command lines within a stream of commands to be skipped during processing of a COMMAND-STREAM. The :SKIP command is particularly useful when used in a :FAIL command, since it enables a number of commands to be skipped when a particular command fails.

:PROMPT command causes a user specified prompting message to be set up. This prompting message displays when a :READ command is issued.

:READ command causes a prompting message, previously set up using a :PROMPT command, to display, and execution of the COMMAND-STREAM to suspend. The user then inputs data in response to the prompt, the data input is held for later use, and execution of the COMMAND-STREAM continues with the next command.

:WRITE command causes data input using the :READ command to be input into the save area below the current line. Data input using a :READ command can be input to the save area as required using the :WRITE command without issuing further :READ commands.

:MEND command must be present in all COMMAND-STREAMs containing Front-end commands and causes execution of a COMMAND-STREAM to end.

Error Messages

As mentioned in the discussion on display areas, error messages from the Interactive Front-end display on the top line of the screen. Error messages which are issued due to the invalid operation of specific commands are listed in the specifications of those commands. The rest of this section lists error messages which are applicable to several commands.

If a command is not one of those provided by the Interactive Front-end, or is not preceded by the PREFIX character, this message issues:

```
*** INVALID COMMAND ***
```

If a parameter obligatory to the command specification is omitted when the command is entered, then this message issues:

```
*** MISSING PARAMETER ***
```

If the value specified for a parameter in the command is invalid (for example, a letter when an integer is required), then this message issues:

```
*** INVALID PARAMETER ***
```

When any of the above messages results from an invalid command operation, the display output remains unchanged.

Command Specifications

This section describes the Command Specifications for interactive front-end commands.

& (Ampersand)

Use the ampersand (&) command to cause the next buffer of output to display, or, use as a prefix to a DesignManager command, to force execution of a DesignManager command before all output in the last output buffer displays.

& Format

```
& [command]
```

where *command* is a valid DesignManager command.

& Remarks

1. If the buffer of output display is not the last buffer of output, then input of the ESCAPE character causes the next buffer of output to display.
2. If the ESCAPE character only is entered and the last buffer of output displays, then the output area clears, and a prompt for the next DesignManager command displays.
3. If the ESCAPE character followed by a DesignManager command is entered and the last buffer of output displays, then the DesignManager command is actioned.
4. If the output buffer display is not the last buffer, then entering the ESCAPE character followed by a DesignManager command causes the next output buffer to display (the DesignManager command is ignored).

5. If the ESCAPE character is entered and the save area buffer displays, then the display switches to the output area and the command is actioned as described in [remark 1 on page 187](#) through [remark 4 on page 187](#) above.

& Examples

```
&  
& LIST ONLY 'USER- ' ;  
& EXECUTE ;
```

:+ (Plus sign)

Use the :+ command to move the screen display down by a specified number of lines.

:+ Format

```
:+n
```

where *n* is an integer equaling number of lines between the line that is currently top of the screen and the line that is to become the current line.

:+ Remarks

1. If the number of lines specified in the command is greater than the number of lines following the current line in the current buffer, then the last screen of the current buffer displays.
2. If the number of lines is not specified in the command, then 1 is assumed.

:+ Example

```
:+5
```

:- (Minus sign)

Use the :- command to move the screen display up by a specified number of lines.

:- Format

```
:- [n]
```

where *n* is an integer equaling number of lines between the line that is currently top of the screen and the line that is to become the current line.

:- Remarks

1. If the number of lines specified in the command is greater than the number of lines preceding the current line in the current buffer, then the first screen of the current buffer displays.
2. If the number of lines is not specified in the command, then 1 is assumed.

:- Examples

```
: -11
```

:ADD

Use the :ADD command to add lines to the save area.

:ADD Format

$$\text{:ADD} \left\{ \begin{array}{l} m \\ \underline{\text{BOTTOM}} \\ \underline{\text{TOP}} \\ * \end{array} \right\} [n]$$

where:

m is the line number of the line after which lines are to be added.

n is the number of lines to add.

:ADD Remarks

1. If the keyword TOP is specified then the new lines add above the lines currently held in the save area.
2. If the keyword BOTTOM is specified then the new lines add below the lines currently held in the save area.
3. If an asterisk (*) is specified then the new lines add below the current line in the save area.
4. If an integer m is specified then the new lines add after line number m in the save area.
5. After entry of the command, the screen display opens up so that the screen displays with the appropriate number of blank lines added. Type in the lines to insert one at a time. When each line is input, the screen redisplay with the new line inserted.
6. Use a null input line to terminate the :ADD command at any time before the number of lines specified has been input.
7. If the number of lines input is sufficient to fit on one screen then the current line remains unaltered. Otherwise, when input reaches a line not on the present screen, the current line is altered to make this line visible. At the end of the command the current line is left where it reached during the command.
8. If the number of lines specified in the :ADD command is greater than the number of lines available in the buffer, then the command rejects with the message:

```
*** INSUFFICIENT ROOM IN BUFFER ***
```

9. If the number of characters input using an :ADD command is greater than the number of character spaces available in the buffer, then the command rejects with the message:

```
*** INSUFFICIENT ROOM IN BUFFER ***
```
10. If the number of lines, *n*, to be added is not specified then the default value assumed is 1.

:ADD Examples

```
:ADD TOP 10
```

```
:ADD 5 6
```

:BOTTOM

Use the :BOTTOM command to display the last screen of the current buffer.

:BOTTOM Format

```
:BOTTOM
```

If the current screen is the last screen of the current buffer, then the command is accepted but has no effect.

:BOTTOM Example

```
:BOTTOM
```

:CHANGE

Use the :CHANGE command to change a character string on specified line(s) to another character string, the position of the current line being unaltered after processing.

:CHANGE Format

```
:CHANGE [F] [s [e]] /string1/string2/  $\left\{ \begin{array}{l} m \\ \underline{B}OTTOM \\ * \end{array} \right\} [ \left\{ \begin{array}{l} n \\ \underline{B}OTTOM \\ * \end{array} \right\} ]$ 
```

where:

s is an integer being the start column number.

e is an integer being the end column number.

string1 is the character string to be changed.

string2 is the new character string to replace *string1*.

m is an integer being the line number of the first line to be changed.

n is an integer being the line number of the last line to be changed.

:CHANGE Remarks

1. If the keyword F is included in the command, then the first occurrence of the character string only changes on each specified line.
2. If the start and end columns are unspecified, then they are assumed to be 1 and 80, respectively. If the end column only is unspecified then it is assumed to be 80. It is not possible to specify the end column number without specifying the start column number.
3. For a character string to change, it must be completely contained within the range of the columns specified.
4. The range of lines on which the :CHANGE command is to operate is specified by the final two operands in the command. If one operand only is included then the single line specified by that operand changes.
5. If a range of lines is specified in the command, then the line specified by the second operand must not be above that specified by the first operand, or the command aborts with the message:

*** INVALID RANGE ***
6. Use the asterisks (*) to specify the current line in the save area.
7. Use the keyword BOTTOM to specify the bottom line in the save area.
8. If a specified string is not found at least once within the area specified then this message issues:

*** DATA NOT FOUND ***
9. If the range of lines specified in the command exceeds the line range contained in the save area, then the command aborts with the message:

*** LINES OUT OF RANGE OF SAVE AREA ***
10. If a line is longer than 80 characters after changing then it is truncated to 80 characters. Any lines truncated in this way display and this message is output:

*** PRESS ENTER TO CONTINUE ***
11. If the command :VERIFY ON is specified then all lines altered display after alteration.
12. Replace the delimiter (/) by any non-alphanumeric character that is not present in the string, however the same character must be used as the delimiter throughout the command.
13. Blank spaces may be included in both string1 and string2.
14. After processing a :CHANGE command, the current line is set to the line which was current before the command was issued.

15. If string2 is not specified, that is if no character string is specified between the delimiters defining the field that normally contains string2, then the effect of the command is to remove occurrences of string1.
16. String1 must always be specified.

:CHANGE Examples

```
:CHANGE F /to/from/BOTTOM
```

```
:CHANGE 1 10 'NAME'NEW-NAME' 1 BOTTOM
```

:COPY

Use the :COPY command to copy lines held in the output area or the save area for later insertion into the save area, using the :INSERT command.

:COPY Format

$$\text{:COPY} \left\{ \begin{array}{l} m \\ \underline{\text{BOTTOM}} \\ * \end{array} \right\} [\left\{ \begin{array}{l} n \\ \underline{\text{BOTTOM}} \\ * \end{array} \right\}]$$

where:

m is an integer being the line number of the first line to be copied.

n is an integer being the line number of the last line to be copied.

:COPY Remarks

1. The first operand in the :COPY command indicates the line at which copying is to begin; the second operand in the command indicates the line at which copying is to end.
2. If one operand only is specified in the :COPY command then the single line indicated by that operand is copied.
3. If two operands are specified in the command, then the line specified by the second operand must not be above that specified by the first operand, or the command aborts with this message:

```
*** INVALID RANGE ***
```
4. Use the keyword BOTTOM to specify the bottom line in the current area.
5. Use an asterisk (*) to specify the current line in the current area.
6. When copying from the save area it is important to note that any changes made to the copied lines before insertion will reflect in the lines when they are inserted.

- If the number of lines set up for insertion into the save area using the :INSERT command, is greater than the number of lines to the end of the buffer then only those lines which fit into the buffer are inserted.

:COPY Examples

```
:COPY * BOTTOM
```

```
:COPY 35
```

:DELETE

Use the :DELETE command to delete specified lines from the save area.

:DELETE Format

$$\text{:DELETE } \left\{ \begin{array}{l} m \\ \text{BOTTOM} \\ * \end{array} \right\} [\left\{ \begin{array}{l} n \\ \text{BOTTOM} \\ * \end{array} \right\}]$$

where:

m is an integer being the line number of the first line to be deleted.

n is an integer being the line number of the last line to be deleted.

:DELETE Remarks

- The first operand in the :DELETE command indicates the first line to delete; the second operand in the command indicates the last line to delete.
- If one operand only is specified in the :DELETE command then the single line indicated by that operand deletes.
- If two operands are specified in the command, then the line specified by the second operand must not be above that specified by the first operand, or the command aborts with the message:


```
*** INVALID RANGE ***
```
- Use the keyword BOTTOM to specify the bottom line.
- Use the asterisks (*) to specify the current line.

:DELETE Example

```
:DELETE 10
```

:DOWN

Use the `:DOWN` command to move the screen display down by a specified number of whole screens so that a particular following screen displays.

:DOWN Format

`:DOWN [n]`

where *n* is an unsigned integer being the number of whole screens between the current screen and the screen to display.

:DOWN Remarks

1. If the number of screens specified in the command is greater than the number of screens following the current screen in the current buffer, then the last screen of the current buffer displays.
2. If the number of screens is not specified in the command, then a default of 1 is assumed. The screen immediately following the current screen in the current buffer displays.

:DOWN Example

`: DOWN 8`

:DUPLICATE

Use the `:DUPLICATE` command to copy a specified number of times, a single line that is already held in the save area.

:DUPLICATE Format

`:DUPLICATE [n]`

where *n* is the number of times the line is to duplicate.

:DUPLICATE Remarks

1. The copies of the duplicated line are added after the current line.
2. The current line before execution of the `:DUPLICATE` command remains the current line after execution of the command.
3. If the number of times a line is to duplicate is not specified then the a default value of 1 is assumed.

:DUPLICATE Example

`:DUPLICATE 5`

:FAIL

Use the :FAIL command to execute a specified command only if the previous command failed.

:FAIL Format

```
:FAIL [command]
```

where *command* is a valid Front-end command.

:FAIL Remarks

1. The :FAIL command is designed for use in a COMMAND-STREAM of Front-end commands: the following remarks apply if it is used in this way. If a single :FAIL command is input during an interactive session, it is accepted but has no effect.
2. If the command immediately previous to a :FAIL command causes an error message then the :FAIL line is executed.
3. When executed the :FAIL command is treated as a normal command. After execution, processing continues with the next command.
4. If no command is specified in the :FAIL command, then execution of the COMMAND-STREAM continues with the next command. This prevents execution of the COMMAND-STREAM from aborting, as would normally be the case if a command in a COMMAND-STREAM causes an error message.
5. If the command specified in the :FAIL command is a :MACRO command, then execution of the :FAIL line causes control to transfer to the COMMAND-STREAM specified in that :MACRO command. Execution terminates on completion of this latter :MACRO command.
6. If the :FAIL command is a macro execution of a non-existent COMMAND-STREAM, then macro execution terminates with the message:

```
*** INVALID COMMAND ***
```

:FAIL Examples

```
:FAIL :SKIP 2
```

```
:FAIL :MACRO MESSAGE1
```

```
:FAIL
```

:INSERT

Use :INSERT to insert into the save area after a specified line, those lines which were previously set up for insertion using either the :COPY command or the :MOVE command.

:INSERT Format

$$\text{:INSERT} \left\{ \begin{array}{l} p \\ \text{BOTTOM} \\ \text{TOP} \\ * \end{array} \right\}$$

where p is an integer being the line number of the line after which lines are to be inserted.

:INSERT Remarks

1. If neither a :COPY command nor a :MOVE command is issued, that is no lines are available for insertion, then no lines insert when the :INSERT command issues.
2. When using in conjunction with the :COPY command, use the :INSERT command to insert the same data a number of times without repetition of the :COPY command.
3. When using in conjunction with the :MOVE command, use the :INSERT command to insert data once only. If an attempt is made to insert again before either a further :MOVE or :COPY command issues, then the command rejects with this message:

```
*** MOVE COMPLETED NO COPY OUTSTANDING ***
```
4. Any subsequent alteration to lines, which have been copied (using either a :COPY or a :MOVE command), are reflected in any insertion of those lines.
5. When used in conjunction with the :COPY command, it is not possible to INSERT within the range of lines copied. If an attempt is made to do so the command rejects with the message:

```
*** CANNOT INSERT WITHIN RANGE OF COPIED LINES ***
```
6. When used in conjunction with the :MOVE command, it is not possible to :INSERT within the range of lines moved. If an attempt is made to do so then the command rejects with the message:

```
*** TARGET WITHIN LINE RANGE TO BE MOVED ***
```
7. If the keyword TOP is specified then the lines insert above the lines currently held in the save area.
8. If the keyword BOTTOM is specified then the lines insert below the lines currently held in the save area.
9. If an asterisks (*) is specified then the lines insert below the current line in the save area.
10. If an integer p is specified then the lines insert after line number p in the save area.

11. If the number of lines inserted is sufficient to fit on one screen then the current line remains unaltered. Otherwise, when the insertion causes overflow onto another screen, the current line alters to make the last line inserted visible. At the end of the command the current line is left at the point it reached during the command.
12. If the number of lines inserted is greater than the number of lines available in the buffer, then the insert carries out until the buffer is full: the remaining lines are not inserted, and this message outputs:

```
*** PRESS ENTER TO CONTINUE ***
```

:INSERT Examples

```
:INSERT TOP
:INSERT *
```

:LOCATE

Use the `:LOCATE` command to locate a specified character string within a specified column range in the current buffer.

:LOCATE Format

```
:LOCATE 'string' [s [e]
```

where:

string is the character string to be located.

s is an integer being the start column number.

e is an integer being the end column number.

:LOCATE Remarks

1. String can be any valid delimited character string, or, if the string consists only of alphanumeric characters and/or hyphens, delimiters can be optionally omitted.
2. If the start and end columns are unspecified then they are assumed to be 1 and 80, respectively. If the end column only is unspecified then it is assumed to be 80. It is not possible to specify the end column number without specifying the start column number.
3. Processing of the command causes the first occurrence of the specified character string after the current line to locate. The line on which the character string is found is then made the current line. Subsequent occurrences of the character string can be located by re-entering the command.
4. If the character string is not located within the range of lines specified, then the current line is unaltered and this message is output:

```
*** DATA NOT FOUND ***
```

5. The start and end column numbers must be unsigned integers in the range 1 through 80. The end column must not be less than the start column number. If an invalid column range is specified, then the command aborts with the message:

```
*** INVALID RANGE ***
```

:LOCATE Examples

```
:LOCATE 'NEW YORK' 50  
  
:LOCATE CUSTOMER-NAME 10 30  
  
:LOCATE 'USERVIEW NAME'
```

:MACRO

Use the :MACRO command to invoke execution of a COMMAND-STREAM member containing Front-end commands.

:MACRO Format

```
:MACRO name
```

where *name* is the name of the COMMAND-STREAM to execute.

:MACRO Remarks

1. If the COMMAND-STREAM specified does not exist then the :MACRO command has no effect.
2. The commands in the COMMAND-STREAM member are taken and executed sequentially until a :MEND command is actioned except as stated in [remark 3](#).
3. If any command in the macro causes an error message to issue then execution of the macro aborts unless the next command is a :FAIL command. If execution aborts this message issues:

```
*** MACRO name ABORTED AT LINE nn ***
```

where:

name is the name of the macro being executed.

nn is the line number of the line containing the erroneous command (counting the first line of commands as line 1).

4. If a COMMAND-STREAM contains a :MACRO command then the COMMAND-STREAM specified in this latter :MACRO command executes as described in [remark 2](#). However, on termination of the latter, COMMAND-STREAM control is not returned to the original COMMAND-STREAM.

5. A COMMAND-STREAM may contain any Front-end commands. It may *not* contain any design or dictionary mode commands. Further, it may *not* contain a command preceded by an escape character.

:MACRO Example

```
:MACRO CHECK1
```

:MEND

Use the :MEND command to define the end of a COMMAND-STREAM member containing Front-end commands.

:MEND Format

```
: MEND
```

:MEND Remarks

1. The :MEND command should be the last command in a COMMAND-STREAM member that contains Front-end commands.
2. If the :MEND command is input as a single command during an interactive session, then the command is accepted but has no effect.

:MEND Example

```
:MEND
```

:MOVE

Use the :MOVE command to move specified lines relative to other lines in the save area.

:MOVE Format

$$\text{:MOVE } \left\{ \begin{array}{l} m \\ \underline{\text{BOTTOM}} \\ * \end{array} \right\} [\left\{ \begin{array}{l} n \\ \underline{\text{BOTTOM}} \\ * \end{array} \right\}] \left\{ \begin{array}{l} p \\ \underline{\text{BOTTOM}} \\ \underline{\text{TOP}} \\ * \\ I \end{array} \right\}$$

where:

m is an integer being the line number of the first line to move.

n is an integer being the line number of the last line to move.

p is an integer being the line number of the target line after which the moved lines are inserted.

:MOVE Remarks

1. Use an asterisks (*) to specify the current line.
2. Use the keyword BOTTOM to specify the last line currently held in the save area.
3. If three operands are specified in the command, then the first two operands specify the range of lines to move, and the final operand specifies the target line after which the moved lines are to be inserted. (See also [remark 5](#) and [remark 8](#).)
4. If two operands only are specified in the command, then the first operand specifies the single line to move, and the second operand specifies the target line after which the moved line is inserted. (See also [remark 5](#) and [remark 8](#).)
5. The keyword TOP provides the capability of inserting moved lines in front of the first line in the save area. Specifying TOP as the target line causes insertion of moved lines before the first line, whereas specifying 1 causes insertion of moved lines after the first line.
6. If three operands are specified in the command, then the line specified by the first operand must not be below that specified by the second operand, or the command aborts with this message:

```
*** INVALID RANGE ***
```
7. The target line must be specified outside of the range of lines moved or the command aborts with this message:

```
*** TARGET LINE WITHIN LINE-RANGE TO BE MOVED ***
```
8. If the final operand in the command is the character I, then insertion of the lines to move suspends until an :INSERT command is issued. The lines insert as defined by the :INSERT command issued (see also [":INSERT" on page 196](#)).

:MOVE Examples

```
:MOVE 10 20 *  
  
:MOVE BOTTOM TOP  
  
:MOVE * BOTTOM I
```

:NUMBERS

Use the :NUMBERS command to turn on or off the numbering of the lines displayed.

:NUMBERS Format

```
:NUMBERS { ON }  
          { OFF }
```

On entry to the Interactive Front-end, numbering is turned off.

:NUMBERS Example

```
:NUMBERS ON
```

:POSITION

Use the :POSITION command to adjust the position of the save area on the screen in order that a particular line becomes the current line.

:POSITION Format

```
:Position n
```

where *n* is an integer being the line number of the line which is to become the current line.

:POSITION Remarks

1. Line numbers display if the command :NUMBERS ON is issued.
2. The position of the new current line relative to the other lines in the save area remains unaltered.

:POSITION Examples

```
:POSITION 15
```

:PROMPT

Use the :PROMPT command to set up a prompt to display when a :READ command is performed.

:PROMPT Format

```
:Prompt message
```

The message may contain any characters including embedded blanks but may not span two lines.

:PROMPT Example

```
:PROMPT INPUT LINE RANGE FOR :CHANGE COMMAND
```

:READ

Use the :READ command to read a parameter to be used in a :WRITE command.

:READ Format

```
:Read
```

:READ Remarks

1. When this command is issued execution suspends and the buffer displays. The line set up by the :PROMPT command displays on the top line of the screen. The :READ command is then performed and the data input is held for later use.
2. The data input for the :READ must not have any embedded blanks.

:READ Example

:READ

:SCHCHANGE

Use the :SCHCHANGE command to change a character string on specified line(s) to another character string, and to set the position of the current line to the line following that last changed.

:SCHCHANGE Format

`:SCHCHANGE [F] [s [e]] /string1/string2/ { m
BOTTOM
* } [{ n
BOTTOM
* }]`

where:

s is an integer being the start column number.

e is an integer being the end column number.

string1 is the character string to change.

string2 is the new character string to replace string 1.

m is an integer being the line number of the first line to change.

n is an integer being the line number of the last line to change.

:SCHCHANGE Remarks

1. If the keyword F is included in the command, then the first occurrence of the character string only, changes on each specified line.
2. If the start and end columns are unspecified, then they are assumed to be 1 and 80, respectively. If the end column only is unspecified then it is assumed to be 80. It is not possible to specify the end column number without specifying the start column number.
3. For a character string to change, it must be completely contained within the range of the columns specified.
4. The range of lines on which the :SCHCHANGE command is to operate is specified by the final two operands in the command. If one operand only is included then the single line specified by that operand changes.

5. If a range of lines is specified in the command then the line specified by the second operand must not be above that specified by the first operand, or the command aborts with the message:

```
*** INVALID RANGE ***
```
6. Use an asterisk (*) to specify the current line in the save area.
7. Use the keyword BOTTOM to specify the bottom line in the save area.
8. If a specified string is not found at least once within the area specified then this message issues:

```
*** DATA NOT FOUND ***
```
9. If the range of lines specified in the command exceeds the line range contained in the save area, then the command aborts with the message:

```
*** LINES OUT OF RANGE OF SAVE AREA ***
```
10. If a line is longer than 80 characters after changing then it is truncated to 80 characters. Any lines truncated in this way display and this message is output:

```
*** PRESS ENTER TO CONTINUE ***
```
11. If the command :VERIFY ON has been specified, then all lines altered display after alteration.
12. The delimiter (/) can be replaced by any non-alphanumeric character that is not present in the string, however the same character must be used as the delimiter throughout the command.
13. Blank spaces may be included in both string1 and string2.
14. If the specified character string is not found the current line is unaltered. If it is found then the current line is set to the line following the last line on which the string was found. If found on the last line then that line becomes the current line.
15. If string2 is not specified, that is if no character string normally contains string2, then the effect of the command is to remove occurrences of string1.
16. Always specify string1.

:SCHANGE Examples

```
:SCHANG F 15 /in/out/ 1 10
```

```
:SCHANG 15 60 'variable'fixed'
```

:SET

Use the :SET command to alter, for the duration of the current interactive session, the existing setting of the ESCAPE character or the PREFIX character.

:SET Format

$$\underline{\text{:SET}} \left\{ \begin{array}{l} \underline{\text{ESCAPE}} \\ \underline{\text{PREFIX}} \end{array} \right\} \textit{character}$$

where *character* is a printable, non-alphanumeric character (a blank space is considered a non-printable character in this context) being a new setting for the PREFIX or the ESCAPE character.

:SET Remarks

1. On entry to the Interactive Front-end, the default settings of ampersand (&) for the ESCAPE character, and colon (:) for the PREFIX character, are taken. If these defaults are altered by tailoring the LOPT1 macro (see ["Overview of Interactive Front-end Commands" on page 182](#)), then these new default settings are taken.
2. Without any parameters, this command causes the current settings of the characters to display at the top of the screen:

```
:SET
```
3. If default values are overridden by issuing :SET commands, messages containing the ESCAPE character and/or the PREFIX character continue to display the default values of these characters (and *not* the values of these characters as set).

:SET Examples

```
:SET PREFIX /  
  
:SET ESCAPE *
```

:SHIFT

Use the :SHIFT command to move specified lines either right or left by a specified number of character positions.

:SHIFT Format

$$\underline{\text{:SHIFT}} \left\{ \begin{array}{l} \underline{\text{L}} \\ \underline{\text{R}} \end{array} \right\} c \left[\left\{ \begin{array}{l} \underline{m} \\ \underline{\text{BOTTOM}} \\ \underline{*} \end{array} \right\} \left[\left\{ \begin{array}{l} \underline{n} \\ \underline{\text{BOTTOM}} \\ \underline{*} \end{array} \right\} \right] \right]$$

where:

c is an integer being the number of character places the line(s) are to move.

m is an integer being the line number of the first line to shift.

n is an integer being the line number of the final line to shift.

:SHIFT Remarks

1. The permissible values of c (number of character positions by which lines are to shift) are 1 through 79. If an attempt is made to shift lines by an invalid amount, then the command aborts with the message:

```
*** INVALID PARAMETER ***
```
2. Any characters shifted past column 80 or to before column 1 are lost and cannot be retrieved by moving back.
3. When moving to the left, the end of the line is padded with blanks. When moving to the right the beginning is similarly padded.
4. The character L causes lines to shift to the left.
5. The character R causes lines to shift to the right.
6. The final two operands specifiable in the command are used to define the lines to shift. The former of these operands indicates the line at which shifting is to begin, and the latter indicates the line at which shifting is to end.
7. If both the final operands are specified, then the line specified by the second operand must not be above that specified by the first operand, or the command aborts with this message:

```
*** INVALID RANGE ***
```
8. If only one of the final two operands is specified in the command, then the single line indicated by that operand shifts.
9. Use the keyword BOTTOM to specify the bottom line in the save area.
10. Use an asterisk (*) to specify the current line in the save area.

:SHIFT Examples

```
:SHIFT L 10
:SHIFT R 2 * 35
```

:SKIP

Use the :SKIP command to cause a specified number of commands in a COMMAND-STREAM of Front-end commands to be skipped.

:SKIP Format

```
SKIP  $n$ 
```

where n is the number of lines in the COMMAND-STREAM to skip.

:SKIP Remarks

1. The :SKIP command is designed for use in a COMMAND-STREAM of Front-end commands: these remarks apply if it is used in this way. If a single :SKIP command is input during an interactive session, it is accepted but has no effect.
2. The :SKIP command is designed particularly for use with the :FAIL command. If it is used in this way a number of commands can be skipped when an erroneous command is encountered.

:SKIP Examples

```
:SKIP 3
```

:SWITCH

Use the :SWITCH command to switch the display between the output area and the save area.

:SWITCH Format

```
:SWITCH
```

The line that is current in a display area when a :SWITCH command executes will be the current line when that area is returned to.

:SWITCH Example

```
:SWITCH
```

:TOP

Use the :TOP command to cause the first screen of the current buffer to display.

:TOP Format

```
:TOP
```

If the current screen is the first screen of the current buffer, then the :TOP command is accepted but has no effect.

:TOP Examples

```
:TOP
```

:VERIFY

Use the :VERIFY command to display lines altered by a :CHANGE or :SCHANG command.

:VERIFY Format

$$\underline{\text{:VERIFY}} \left\{ \begin{array}{l} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{array} \right\}$$

When verification is on, all lines altered by a :CHANGE or :SCHANG command display after alteration.

:VERIFY Examples

```
:VERIFY ON
```

```
:VERIFY OFF
```

:UP

Use the :UP command to move the screen display up by a specified number of whole screens so that a particular preceding screen displays.

:UP Format

$$\underline{\text{:UP}} \ [n]$$

where n is an unsigned integer being the number of whole screens between the current screen and the screen to display.

:UP Remarks

1. If the number of screens specified in the command is greater than the number of screens preceding the current screen in the current buffer, then the first screen of the current buffer displays.
2. If the number of screens is not specified in the command, then a default of 1 is assumed. The screen immediately preceding the current screen in the current buffer then displays.

:UP Example

```
:UP 5
```

:WIPE

The :WIPE command is used to clear the save area.

:WIPE Format

$$\underline{\text{:WIPE}}$$

Input of a :WIPE command causes the save area to completely clear of data so that it is available for new usage.

:WIPE Example

`:WIPE`

:WRITE

Use the `:WRITE` command to add to the save area the data input previously using a `:READ` command.

:WRITE Format

`:WRITE`

:WRITE Remarks

1. The data read in is added to the save area immediately after the current line.
2. If no `:READ` command performs then the `:WRITE` command has no effect.
3. The data read in by one `:READ` command is available for use by successive `:WRITE` commands without the need to input another `:READ` command.

:WRITE Example

`:WRITE`

:EXECUTE

Use the `:EXECUTE` command to submit the contents of the save area as a command or a stream of commands to DesignManager for execution.

:EXECUTE Format

`EXECUTE { ; }`
`{ . }`

:EXECUTE Remarks

1. The `:EXECUTE` command is a design mode command, and thus must be used in conjunction with the ESCAPE character (&) during an interactive session under the Front-end.
2. When the `:EXECUTE` command is issued the data in the save area must be set up as a valid DesignManager command or stream of commands. All DesignManager design and dictionary mode commands are considered valid commands in this context; Front-end commands including the `:EXECUTE` command are considered invalid.

:EXECUTE Example

`EXECUTE`

10

User Printer Graphics (DSR-UD31)

Introduction

The User Printer Graphics optional additional facility (selectable unit DSR-UD31) provides the capability of obtaining a detailed analysis of the relations held in the workbench design area.

As the name User Printer Graphics implies, the analysis is output in an easily read graphic plot; details of the content and layout of the plots available are given in "[Logical Schema Cluster Plot](#)" on page 210 and "[Network Cluster Plot](#)" on page 214. If the User Formatted Output facility (selectable unit DSR-UD30) is also installed, the output of the User Printer Graphics facility may be user-formatted.

Command Specifications

PLOT

Use the PLOT command to produce a diagrammatic representation of the relations in the workbench design area according to a specified format selection.

PLOT Format

```
PLOT { LOGICAL-SCHEMA } [format-selection] { ; }  
      { NETWORK          } { . }
```

where *format-selection* is:

```
{ DETAILS }  
{ SUMMARY }
```

PLOT Remarks

1. If there are no relations in the workbench design area when the PLOT command is issued (that is, the workbench design area is empty or contains unnormalized data), the command is invalid, is reported as such, and processing continues with the next command.
2. If format-selection is not specified, DesignManager defaults to SUMMARY.
3. The options in the format-selection clause are extended if the User Formatted Output facility (DSR-UD30) is installed.

PLOT Examples

```
PLOT NETWORK DETAILS ;  
  
PLOT LOGICAL-SCHEMA SUMMARY ;
```

Logical Schema Cluster Plot

The Logical Schema Cluster Plot gives a diagrammatic representation of the direct foreign key associations and the direct hierarchical associations (see ["Glossary" on page 245](#)) between the relations held in the workbench design area. The plot consists of these two parts:

- An individual logical schema cluster (see [Figure 7 on page 213](#)) for each relation generated by the DESIGN command, in the workbench design area.
- An association matrix (as described in ["The Logical Schema Association Matrix" on page 212](#)).

The Individual Logical Schema Cluster

An individual logical schema cluster is plotted for each relation in the workbench design area which the DESIGN command generates. Throughout this section the relation for which the individual logical schema cluster is plotted is termed the subject relation.

The outline in [Figure 7 on page 213](#) gives an example of every type of association possible in a logical schema cluster between a subject relation and its directly associated relations. The type of association between the subject relation and the associated relation is indicated in the cluster by the type of arrow used. The meaning of each type of arrow is given in [Figure 8 on page 214](#), which therefore also defines the associations plotted in a Logical Schema Cluster Plot.

Note: _____

Except for the arrow representing the association with R4, which is single-headed in both directions, the relations in the plot are arranged so that a single-headed arrow points upward and a double-headed arrow points downward.

Two categories of Logical Schema Cluster Plot are available using the PLOT command: summary plot and detail plot. The standard form of the plot is the same for both categories; however the detail in which the relations are reported is greater in the case of the detail plot. Additionally the plot can be user-formatted if the User Formatted Output facility (selectable unit DSR-UD30) is installed.

This information is given for the subject relation in a summary plot:

- The relation number; the number allocated to the relation when it generates in the workbench design area
- The relation name (if any)

The information given for the subject relation in a detail plot is as for the summary plot and, in addition, includes these elements:

- The data elements forming the key of the relation
- The non-key data elements in the relation

This information is given for each associated relation in a summary plot:

- The relation number
- The relation name (if any)
- The direct association between the subject relation and the associated relation. The type of association is indicated by the type of arrow used in the plot. The meaning of each type of arrow is defined in [Figure 8 on page 214](#).

The information given for each associated relation in a detail plot is as for the summary plot and, in addition the key data elements forming the key of the relation.

The Logical Schema Association Matrix

When the individual logical schema clusters for all the relations (generated by the DESIGN command) in the workbench design area have been produced the association matrix is output. The logical schema association matrix is a tabular summary showing all the direct associations between all the relations in the workbench design area. Each relation is shown by its relation number on both the vertical *FROM* axis of the matrix and the horizontal *TO* axis of the matrix. By reading across the matrix it is possible to determine all the direct associations from a selected relation to other relations. Alternatively, by reading down the matrix it is possible to determine all the direct associations to a selected relation.

If there is no direct association from a selected relation to a particular relation then this position is left blank in the matrix. These two characters indicate the existence of an association from one relation to another:

- 1, which is equivalent to a single arrowhead in the individual logical schema cluster (see [Figure 8 on page 214](#)).
- M, which is equivalent to a double arrowhead in the individual logical schema cluster.

Use the association matrix as a quick reference not only to determine the existence of an association between relations but to ascertain the association type.

Figure 7. Outline of Individual Logical Schema Cluster

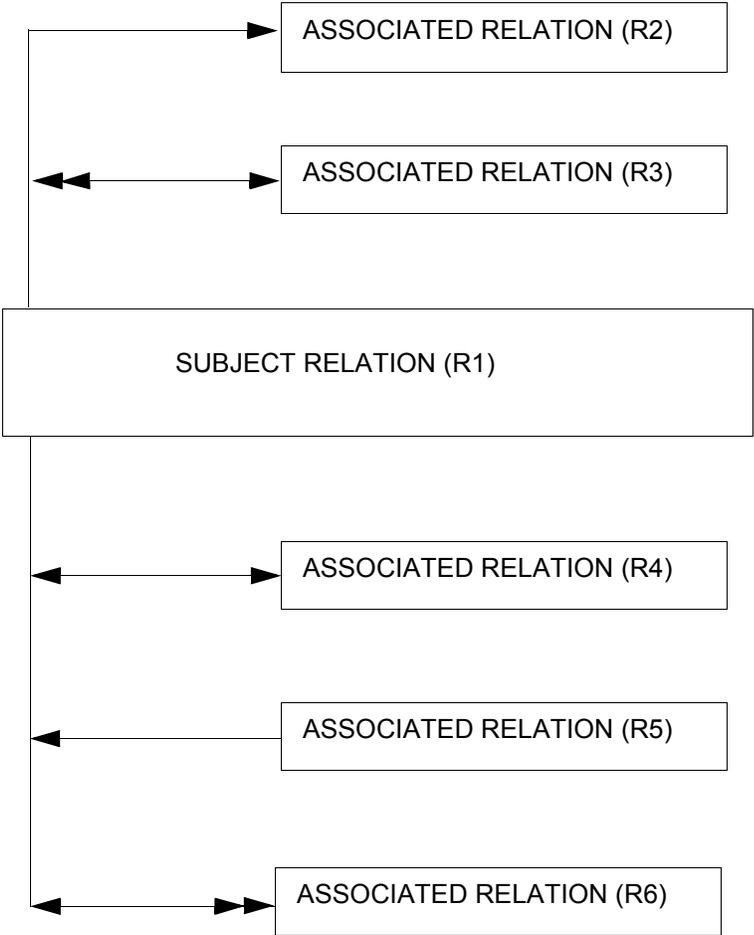


Figure 8. Meaning of Associations Plotted in the Logical Schema Cluster Plot

| Type of Arrow | Meaning of Association |
|---------------|---|
| R1 -----> R2 | A direct foreign key association (see "Glossary" on page 245) exists from the subject relation (R1) to the associated relation (R2). For example, R1(A,B) with key A, and R2(B,C) with key B. |
| R1 <<----> R3 | A direct foreign key association (see "Glossary" on page 245) exists from the subject relation (R1) and the associated relation (R3), such that the key of R3 is contained in the key of R1. For example, R1(A,B,E) with key A+B, and R3(A,D) with key A. |
| R1 <-----> R4 | A direct foreign key association exists from the subject relation (R1) to the associated relation (R4), and a direct foreign key association exists from R4 to R1. For example, R1(A,B) with key A, and R4(B,A,C) with key B. |
| R1 <----- R5 | A direct foreign key association exists from the associated relation (R5) to the subject relation (R1). For example, R1(A,B) with key A, and R5(C,A) with key C. |
| R1 <---->> R6 | A direct hierarchical association exists between the subject relation (R1) and the associated relation (R6), such that the key of R1 is contained in the key of R6. For example, R1(A,D) with key A, and R6(A,B,E) with key A+B. |

Network Cluster Plot

The Network Cluster Plot gives a comprehensive diagrammatic representation of the associations between relations in the workbench design area. The relations plotted include all existing FD-relations held in the workbench design area together with those assumed relations added to the workbench design area (as described in ["Creation of Assumed Relations" on page 215](#)). The plot consists of these two parts:

- For each FD-relation and assumed relation, an individual network cluster (see [Figure 9 on page 218](#)) together with details of the secondary key associations (see ["Glossary" on page 245](#)) in that cluster.
- Association matrices. Two association matrices are output at the end of the plot, a hierarchical association matrix and a non-hierarchical association matrix. ["The Network Cluster Association Matrices" on page 217](#) describes these matrices.

Creation of Assumed Relations

These are the relations plotted in a Network Cluster Plot:

- The FD-relations already present in the workbench design area
- The assumed relations created via the MVDs held in the workbench design area

Assumed relations are created and added to the workbench design area in these circumstances:

- If there is an MVD in the workbench design area whose right-hand side (when extraneous data elements have been removed) is not a key in any existing FD-relation, or assumed relation. For example, if there is a dependency $A \text{ ---->> } B+C$ and there is no existing relation which has $B + C$ as a key, then an assumed relation is added to the workbench design area. This assumed relation has $B+C$ as its key.
- If there is an MVD in the workbench design area whose left-hand side (when extraneous data elements are removed) is not contained in any existing FD-relation or assumed relation. For example, if there is a dependency $A+B \text{ ---->> } C$ and there is no existing relation which contains both A and B , then an assumed relation is added to the workbench design area. This assumed relation has $A+B$ as its key.

Individual Network Clusters

An individual network cluster is plotted for each FD-relation and assumed relation in the workbench design area. Throughout this section the relation for which the individual network cluster is being plotted is termed the subject relation.

[Figure 9 on page 218](#) shows the outline of an individual network cluster. The type of association between the subject relation and the associated relation is indicated in the cluster by the type of arrow used and the position in which that associated relation is output relative to the subject relation. Each of the associated relations shown in [Figure 9 on page 218](#) has a different association with a subject relation. [Figure 10 on page 219](#) gives the exact meaning of the associations shown in [Figure 9 on page 218](#), and therefore defines the associations which are plotted in a Network Cluster Plot.

Two categories of Network Cluster Plot are available using the PLOT command: summary plot and detail plot. The standard form of the plot is the same for both categories; however, the detail in which the relations are reported is greater in the case of the detail plot. Additionally, the plot can be user-formatted if the User Formatted Output facility (selectable unit DSR-UD30) is installed.

- The relation number; the number allocated to the relation when it was generated in the workbench design area.
- The relation name (if any), or the word ASSUMED if the subject relation is an assumed relation.

The information given for the subject relation in a detail plot is as for the summary plot and, in addition, includes these elements:

- The data elements forming the key of the relation
- The non-key data elements in the relation

This information is given for each associated relation in a summary plot:

- The relation number
- The relation name (if any), or the word ASSUMED if the associated relation is an assumed relation
- The association between the subject relation and the associated relation

The type of association is indicated by the type of arrow used in the plot. The meaning of each type of arrow is defined in [Figure 10 on page 219](#).

The information given for each associated relation in a detail plot is as for the summary plot and in addition, the data elements forming the key of the associated relation.

Each individual network cluster is followed by information on the secondary key associations contained within it. The information is output in these two parts:

- Summary totals of the secondary keys associated with the subject relation. These consist of:
 - The total number of secondary key associations leading into the subject relation (shown as associated relation R12 in [Figure 9 on page 218](#)).
 - The total number of secondary key associations leading from the subject relation (shown as associated relation R13 in [Figure 9 on page 218](#)).
- For each secondary key association in the cluster, information on its origin is given. The information output is dependent on the category of plot (summary or detail) produced, and is described below.

The information output for each secondary key association in a summary plot is:

- The relation number of the relation from which the secondary key association leads, followed by the relation number of the relation to which the association leads. These relation numbers are those of the subject relation and the associated relation and are output in the order determined by the type of secondary key association.
- The absolute dependency number of the MVD which caused the secondary key association.
- The data elements forming the left-hand side of the MVD which caused the secondary key association.

The information output for each secondary key association in a detail report is as for the summary report and in addition, includes this information:

- The relation name (if any) or the keyword ASSUMED (if the relation is an assumed relation) for the subject relation and for the associated relation.
- For the MVD which caused the secondary key association:
 - The data elements forming its right-hand side
 - Its relative dependency number
 - The name of the userview from which it originated

The Network Cluster Association Matrices

When the individual network clusters for all the FD-relations and assumed relations in the workbench design area have been produced, the association matrices are output. Each relation is shown by its relation number on both the vertical *FROM* axis of the matrix, and the horizontal *TO* axis of the matrix. These two association matrices are output:

- The hierarchical association matrix
- The non-hierarchical association matrix

The hierarchical association matrix gives a tabular summary of the relations (both assumed and FD-relations) in the workbench design area. Direct hierarchical associations are shown as associated relations R10 and R11 in [Figure 9 on page 218](#).

If no hierarchical association exists from a selected relation to a particular relation, then this position is left blank in the matrix. However if a direct hierarchical association exists between the two relations, one of these characters is output:

- 1, which is equivalent to the single arrowhead shown with associated relations R10 and R11 in the individual network cluster in [Figure 9 on page 218](#).
- M, which is equivalent to the double arrowhead shown with associated relations R10 and R11 in the individual network cluster in [Figure 9 on page 218](#).

If an association of type 1 (as defined above) is shown in the hierarchical matrix from, for example relation 6 to relation 2, then there must be an association of type M from relation 2 to relation 6 (which is also shown in the matrix).

The non-hierarchical association matrix gives a tabular summary of the direct foreign key associations and the multivalued associations between the relations (both assumed and FD-relations) in the workbench design area. The non-hierarchical associations included in the matrix are shown as associated relations R2 to R9 inclusive in [Figure 9 on page 218](#). Secondary key associations are not included in this matrix because more than one secondary key association may exist from one relation to another. Therefore, secondary key associations are detailed separately as part of the individual network cluster output.

If no non-hierarchical association (as defined above) exists from a selected relation to a particular relation, then this position is left blank in the matrix. However, if a non-hierarchical association exists between the two relations, one of these characters is output:

- 1, which is equivalent to the single arrowhead shown with associated relations R2 through R9 in the individual network cluster in [Figure 9 on page 218](#).
- M, which is equivalent to the double arrowhead shown with associated relations R2 through R9 in the individual network cluster in [Figure 9 on page 218](#).

Figure 9. Outline of Individual Network Cluster Showing Possible Associations



Figure 10. Meaning of Associations Plotted in the Network Cluster Plot

| Type of Arrow | Meaning of Arrow |
|----------------|--|
| R1 <<----> R2 | <p>A direct foreign key association (see "Glossary" on page 245) exists from the subject relation (R1) to the associated relation (R2), and a multivalued association (see "Glossary" on page 245) holds from R2 to R1.</p> <p>For example, R1(A,B,C) with key A, R2 (B,C,D) with key B+C and there is an MVD from B+C to A.</p> |
| R1 -----> R3 | <p>A direct foreign key association (see "Glossary" on page 245) exists from the subject relation (R1) to the associated relation (R3).</p> <p>For example, R1(A,B) with key A, and R3(B,C) with key B.</p> |
| R1 <<----- R4 | <p>A multivalued association (see "Glossary" on page 245) holds from the associated relation (R4) to the subject relation (R1).</p> <p>For example, R1(A,B) with key A, R4(C,D) with key C and there is an MVD from C to A.</p> |
| R1 <<-->> R5 | <p>Multivalued associations hold both from the subject relation (R1) to the associated relation (R5), and from R5 to R1.</p> <p>For example, R1(A,B) with key A, R5(C,D) with key C and there are MVDs from A to C and from C to A.</p> |
| R1 <-----> R6 | <p>Direct foreign key associations exist both from the subject relation (R1) to the associated relation (R6) and from R6 to R1.</p> <p>For example, R1(A,B) with key A, and R6(B,A,C) with key B.</p> |
| R1 <---->> R7 | <p>A direct foreign key association exists from the associated relation (R7) to the subject relation (R1), and a multivalued association holds from R1 to R7.</p> <p>For example, R1(A,B) with key A, R7(C,A,D) with key C and there is also an MVD from A to C.</p> |
| R1 <----- R8 | <p>A direct foreign key association exists from the associated relation (R8) to the subject relation (R1).</p> <p>For example R1(A,B) with key A, and R8(C,A) with key C.</p> |
| R1 ----->> R9 | <p>A multivalued association holds from the subject relation(R1) to the associated relation (R9).</p> <p>For example R1(A,B) with key A, R9(C,D) with key C and there is an MVD from A to C.</p> |
| R1 <<----> R10 | <p>A direct hierarchical association (see "Glossary" on page 245) exists between the subject relation (R1) and the associated relation (R10), such that the key of R10 is contained in the key of R1.</p> <p>For example, R1(A,B,E) with key A+B, and R10(A,D) with key A.</p> |

| Type of Arrow | Meaning of Arrow |
|----------------|---|
| R1 <----> R11 | <p>A direct hierarchical association exists between the subject relation (R1) and the associated relation (R11), such that the key of R1 is contained in the key of R11.</p> <p>For example, R1(A,D) with key <i>A</i>, and R11(A,B,E) with key <i>A+B</i>.</p> |
| R12 ----> R1 | <p>A secondary key association (see "Glossary" on page 245) holds from the associated relation (R12) to the subject relation (R1).</p> <p>For example R12(C,D) with key <i>C</i>, R1(A,B) with key <i>A</i>, and there is an MVD from <i>D</i> to <i>A</i>.</p> |
| R13 <<----- R1 | <p>A secondary key association holds from the subject relation (R1) to the associated relation (R13)</p> <p>For example, R13(C,D) with key <i>C</i>, R1(A,B) with key <i>A</i>, and there is an MVD from <i>B</i> to <i>C</i>.</p> |

Appendix

Technical Background

All reference numbers given in this appendix correspond to the references listed in the ["Bibliography" on page 257](#).

The design procedure used in DesignManager generates a third normal form (3NF) relational schema consisting of 3NF relations defined over the input data elements. The schema is produced using the *synthesis* approach of Bernstein (see [reference 1 on page 257](#) and [reference 3 on page 257](#)), extended to permit specification of multivalued dependencies (MVDs) as well as functional dependencies (FDs). Briefly, given a set of data elements and a set of FDs and/or MVDs holding amongst the data elements, the DesignManager procedure removes redundant FDs from the given set and generates (synthesizes) a set of 3NF relations from the remaining dependencies.

In contrast to the synthesis approach used by DesignManager, the alternative approach to schema design is *decomposition* of input relations. In the decomposition approach, a set of one or more relations is given, each consisting of a set of data elements and each with an associated set of dependencies. The given relations need not be in 3NF or even in second normal form (2NF). Using the dependency information, the relations are normalized through successive decomposition into smaller relations. (This is done in such a way that preserves the data content of the input relations. That is, for all instances of the input relations, natural joins of the output relations can be obtained which reproduce the data content of the input relations without loss of information. Such a schema is said to have the *lossless join* property; see [reference 4 on page 257](#).) If the given dependencies are all FDs, the decomposition approach normally used (due to Codd; see [reference 6 on page 257](#)) results in a 3NF relational schema. If the given dependencies include MVDs, the approach used (due to Fagin; see [reference 7 on page 257](#) and [reference 9 on page 258](#)) yields relations in fourth normal form (4NF). In the Fagin procedure, only non-trivial Fagin-MVDs can be used to decompose relations (see ["The Normal Forms Defined" on page 230](#)).

Note: _____

If the Fagin rather than the Codd procedure is used when only FDs are given, the resulting relations will be in Boyce-Codd normal form (BCNF), which is stronger than 3NF (see ["The Normal Forms Defined" on page 230](#)).

Both approaches to schema design have their advantages and disadvantages. The obvious advantage of decomposition is that it can produce 4NF relations. A major disadvantage, if the input includes MVDs, is that either:

- The MVDs must all be restricted to Fagin-MVDs which is very difficult to determine at input time (especially if only a single relation is input, containing all the input data elements).
- Each time an MVD is used to decompose a relation, in order to be sure that no data content information is lost, it must be determined interactively that the MVD is in fact a Fagin-MVD, which can be a slow and cumbersome process.

The major overriding advantage of the DesignManager extended synthesis procedure is that, although it does not achieve 4NF, it is completely automated and does not require the user to determine which of the MVDs are Fagin-MVDs. (After producing 3NF relations via the synthesis approach, it would then be possible to employ an interactive procedure to decompose MVD-relations on the basis of non-trivial Fagin-MVDs. As indicated in ["DSR-MVD Versus Fagin-MVD" on page 243](#), it would be much simpler to decide which MVDs are Fagin-MVDs at this point than at input time when there are no available relations.)

A second major advantage of the synthesis approach is its compatibility with the principle of *faithful representation* of input dependencies. That is, the generated 3NF relations represent, either directly or indirectly, *all* of the input dependencies whether or not any have been altered or eliminated as redundant during the normalization process. Stated another way, each input userview is satisfied by the generated schema. This is not necessarily true for a schema obtained by decomposition. Indeed, it can be shown that it is impossible in all cases to produce a 4NF schema, or even a BCNF schema, and still satisfy the faithful representation property (or the stronger *faithful closure* property satisfied by the DesignManager procedure; see [reference 4 on page 257](#)).

The remaining sections include discussion of relations, schemas, dependencies, derivability and redundancy of dependencies, keys, the various normal forms, and the DesignManager design procedure. Finally, some of the advantages are discussed of specifying DSR-MVDs rather than Fagin-MVDs as input to the design procedure (the former being the DesignManager version of the MVD, defined in ["Relations and Dependencies" on page 223](#)). An effort has been made to present a fairly self-contained discussion. Many of the pertinent definitions appearing in the ["Glossary" on page 245](#), and some of the material presented in [Chapter 6, "DesignManager Procedures and Use," on page 123](#) are repeated here for the sake of completeness.

The Relational Model of a Database

This section describes relations and dependencies, rules of inference for deriving additional FDs, and keys and representation of FDs in a schema.

Relations and Dependencies

A relational database model is a set of *relations* of time-varying data content defined over a set of data elements. A relation R with the set $\{A_1, A_2, \dots, A_n\}$ of data elements is written $R(A_1, A_2, \dots, A_n)$. An *instance* of R , that is, its data content at any instant of time, can be visualized as a two-dimensional table of values. Each column of the table represents a data element of R and consists of values from the domain of that data element. Each row is a set of values, one per data element, where no two rows are identical. The rows of the table constitute the elements of the *R-instance* and usually vary in number and content with time as values are entered, updated, and deleted. The structure of a relation R , on the other hand, called the *relation scheme*, is more or less static, consisting of a list of its data elements and a set of dependencies which hold amongst them. A *relational schema* is simply a set of relation schemes. The dependencies of a relation scheme are in the form of *functional dependencies* (FDs) and *multivalued dependencies* (MVDs). In the DesignManager synthesis approach, FDs and MVDs do not require the context of a relation for their existence (indeed, relations are generated from dependencies) and so are defined separately. However, dependencies are *represented* (see ["Keys and Representation of FDs in a Schema" on page 227](#)) by the relations they produce, and they are said to be *valid* for all instances of the relations.

An FD is a statement of this form where X and Y are sets of one or more data elements and need not be distinct:

$$X \text{ -----} \rightarrow Y$$

It states that each X -value (that is, set of values of the data elements in X) determines exactly one Y -value, Y is said to be *functionally dependent* on X .

If X and Y are subsets of the data elements of a relation R , then an instance of R is said to *obey* the FD:

$$X \text{ -----} \rightarrow Y$$

if each pair of its rows having identical X -values also have identical Y -values (if X is a key of R , see ["Keys and Representation of FDs in a Schema" on page 227](#), or contains a key of R , then no instance of R can have more than one row with a given X -value).

This FD is *valid* in R if every instance of R obeys the FD (the FD is also said to *hold* in R):

$$X \text{ -----} \rightarrow Y$$

A set F of FDs is valid in R if every FD in F is valid in R .

An MVD where X and Y are sets of data elements, states that a variable number of Y -values (perhaps none) is determined by each X -value:

$$X \text{ -----} \twoheadrightarrow Y$$

Y is said to be *multiply dependent* on X. This definition of an MVD (or *DSR-MVD*) is less restrictive and considerably easier for the user than the definition used by Fagin (see "[The Normal Forms Defined](#)" on page 230 and [reference 7](#)). The differences between the two definitions and the justification for the definition used here are described in "[DSR-MVD Versus Fagin-MVD](#)" on page 243 and in the "[Glossary](#)" on page 245 under *Multivalued Dependency as Defined by R. Fagin*. Indeed, the above definition is sufficiently free from restriction that formally the DSR-MVD:

$X \text{ ---->> } Y$

is valid in every relation R whose set of data elements includes both X and Y as subsets (Every FD is a special case of an MVD). That is, for every instance of R, if any two rows have identical X-values, the corresponding Y-values may be the same or they may be different; moreover, some (or all) of the corresponding data elements in Y may have no value whatever defined.

The above definitions are summarized here:

- The FD $X \text{ ----> } Y$ states that, at any instant of time, a unique Y-value is associated with each X-value.
- The MVD $X \text{ ---->> } Y$ states that, at any instant of time, a unique set of Y-values is associated with each X-value (this, of course, is true of a Fagin-MVD too, but it is not enough to characterize a Fagin-MVD).

To illustrate the definitions of FD and MVD, consider the relation R(EMPLOYEE,ADDRESS,AGE,CHILD). The instance of R shown in [Figure 11 on page 224](#), obeys these dependencies:

EMPLOYEE ----> ADDRESS

EMPLOYEE ----> AGE

EMPLOYEE ---->> CHILD

Figure 11. Instance of Relation R Illustrating the Definitions of FD and MVD

| R | | | |
|----------|------------------|-----|-------|
| EMPLOYEE | ADDRESS | AGE | CHILD |
| Adams | 3 N. F Street | 48 | Paul |
| Adams | 3 N. F Street | 48 | Jane |
| Adams | 3 N. F Street | 48 | Anne |
| Adams | 3 N. F Street | 48 | Fred |
| Baker | 22 S. M Street | 24 | - |
| Chase | 17 E. 2nd Avenue | 37 | Alan |
| Davis | 29 W. 5th Avenue | 50 | Mary |

It is evident (from semantic considerations) that every instance of R would obey the given dependencies so that they are valid in R. These remarks are pertinent to the above example (see ["Glossary" on page 245](#) and ["The Normal Forms Defined" on page 230](#) for undefined terms):

Remark 1. R is an example of an unnormalized relation; it would not be produced by DesignManager from the given dependencies. The DesignManager synthesis procedure would generate this third normal form relations instead:

R1 (EMPLOYEE, ADDRESS, AGE)
R2 (EMPLOYEE, CHILD)

R1 and R2 are projections of R on the indicated subsets of R's data elements. Instances of R1 and R2 corresponding to the given R-instance are shown in ["Relations and Keys" on page 134](#).

Remark 2. The MVD EMPLOYEE \twoheadrightarrow CHILD is a Fagin-MVD as well as being a DSR-MVD. For the given instances, the join of R1 and R2 is R, that is, $R = R1 * R2$. Stated another way, the set {ADDRESS, AGE} of data elements and the data element CHILD are independent in R with respect to the data element EMPLOYEE. That is, for each value of EMPLOYEE in any R-instance, there are as many rows as there are combinations of {ADDRESS, AGE} values and values of CHILD. So, there are four Adams rows in the given R-instance and one row each for Baker, Chase, and Davis.

Rules of Inference for Deriving Additional FDs

In the synthesis approach to schema design, only dependencies are given, not relations. As a consequence, the normalization process (see ["The Extended Synthesis Procedure for Schema Design" on page 238](#)) is concerned with determining which of the given FDs are *redundant* and eliminating them before generating relations. An FD is *redundant* in a given set of FDs if it is *implied* by the other FDs of the set. In general, an FD, f , is *implied* by a set F of FDs if f is valid in every relation in which the set F is valid. That is, no instance of any relation can exist which obeys all the FDs of F but which does not obey f . (Every FD belonging to the set F satisfies the definition trivially and so is implied by F .) Therefore, an FD, f , is redundant in a set F of FDs if f is implied by the reduced set $F - \{f\}$.

To determine the set of FDs implied by a given set F , a system of *inference rules* is used, which enables the formal derivation of additional FDs, in a finite number of steps, from the given set. The *closure* F^+ of F is F itself plus all possible FDs derivable from F using the inference rules. A system of inference rules for FDs is *complete* if, for any set F of FDs, the set F^+ is exactly the set of FDs implied by F .

What follows is a complete system of inference rules for FDs (see [reference 1 on page 257](#), [reference 2 on page 257](#), and [reference 11 on page 258](#)) with examples. In the rules, X , W , Y , and Z are sets of data elements which are not necessarily disjoint. In the examples, $A1$ and $A2$ are data elements, and the symbol \emptyset denotes the null set. Throughout, the plus (+) sign is used to denote the union operator between two sets of data elements ($X + Y$, for example, is the union of the sets X and Y).

Rule FD1 (Reflexivity):

If Y is a subset of X or is equal to X (written $Y \subseteq X$), then $X \twoheadrightarrow Y$.

Note:

This is referred to as a *trivial* FD, one which is valid in every relation whose set of data elements includes X as a subset.

Examples:

$A_1 \twoheadrightarrow A_1$

and

$A_1 + A_2 \twoheadrightarrow A_1$

Rule FD2 (Augmentation):

If $X \twoheadrightarrow Y$ and $Z \subseteq W$, then $X+W \twoheadrightarrow Y+Z$

Examples:

$X \twoheadrightarrow Y$ implies $X+A_1 \twoheadrightarrow Y+A_1$ (letting $Z = W = A_1$)

$X \twoheadrightarrow Y$ implies $X+A_1+A_2 \twoheadrightarrow Y+A_1$ (where $Z = A_1, W = A_1+A_2$)

$X \twoheadrightarrow Y$ implies $X+A_1 \twoheadrightarrow Y$ ($Z = \emptyset, W = A_1$)

Rule FD3 (Pseudotransitivity):

If $X \twoheadrightarrow Y$ and $Y+W \twoheadrightarrow Z$, then $X+W \twoheadrightarrow Z$ (For $W = \emptyset$, this reduces to simple transitivity.)

Example:

$X \twoheadrightarrow Y$ and $X+Y \twoheadrightarrow Z$ imply $X \twoheadrightarrow Z$ (letting $W = X$)

(This is one of the generic examples of eliminating an extraneous data element from the left-hand side of an FD; see ["The Extended Synthesis Procedure for Schema Design" on page 238](#)).

Although this system of rules is complete, other useful rules can be derived from them. Permitting, without loss of generality, the assumption that all FDs are *elemental*, that is, each has only a single data element on the right-hand side.

Rule FD4 (Union):

If $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow Y+Z$

Rule FD5 (Decomposition):

If $X \twoheadrightarrow Y+Z$, then $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$

Remark 1. Although the system of rules, FD1, FD2, FD3, is complete, there are alternative systems which are complete, for example, FD1, FD3, FD5.

Remark 2. The given system, FD1, FD2, FD3, is not minimally complete; it contains a subsystem, FD1, FD3, which is also complete. In particular, FD2 is derivable from FD1 and FD3; that is, given that:

$$Z \subseteq W \text{ and } X \text{ -----} > Z,$$

It can be proven, using FD1 and FD3, that:

$$X+W \text{ -----} > Y+Z$$

Proof:

1. $Z \subseteq W$ (given)
2. $Y+Z \subseteq Y+W$ (from (1) by elementary set theory)
3. $X \text{ -----} > Y$ (given)
4. $Y+W \text{ -----} > Y+Z$ (from (2) by FD1)
5. $X+W \text{ -----} > Y+Z$ (from (3) and (4) by FD3)

As indicated above, the normalization process, given a set F of FDs, requires the elimination of redundant FDs (if any exist) from the set F . We see now that an FD, f in F , is redundant in F if it is in the closure of the set obtained by removing f from F , that is, if f is in $(F-\{f\})^+$. In this case, $F-\{f\}$ is a subset of F with the same closure as F and therefore can be used to replace F . In general, any set of FDs with the same closure as F is called a *covering* of F (it need not be a subset of F). A covering is *redundant* if it contains any redundant FDs. Thus the normalization procedure of ["The Extended Synthesis Procedure for Schema Design" on page 238](#) produces a nonredundant covering of F which is used to generate FD-relations (MVD-relations are generated from the given MVDs).

In the DesignManager normalization procedure, redundancy of an FD is determined using a *fast* (linear-time) algorithm, due to Beeri and Bernstein (see [reference 1 on page 257](#)). For a given FD, f , and a set, F , of FDs, the algorithm solves the so-called *membership* problem, that of determining whether or not f is in the closure of F .

Keys and Representation of FDs in a Schema

Let R be a relation with data element set T (that is, $T = \{A_1, A_2, \dots, A_n\}$). A subset $X \subseteq T$ is a *key* of R if T is functionally dependent on X but not on any proper subset of X . Thus X is a key of R if every data element of R is functionally dependent on X and no subset of X has the same property (recall that every data element of X is trivially functionally dependent on X by inference rule FD1 of ["Rules of Inference for Deriving Additional FDs" on page 225](#)). In application, a key of R is any minimal set X of the data elements of R such that, in every instance of R , each X -value determines exactly one row of the instance.

As a consequence of the definition, it is seen that every relation (whether or not any FDs are specified for it) must have at least one key. For, if no other set qualifies, then the entire set of data elements must be the key. Such a relation is termed *all-key*. If R has more than one key designated, each is referred to as an *alternative* (or *candidate*) *key* of R. If a key contains more than one data element, it is called *composite* (or *concatenated*).

In the illustration of ["Relations and Dependencies" on page 223](#), for example, the data element EMPLOYEE is the key of R1(EMPLOYEE,ADDRESS,AGE). EMPLOYEE is not, however, a key of either R2(EMPLOYEE,CHILD) or R(EMPLOYEE,ADDRESS,AGE,CHILD). Both relations instead have the composite key EMPLOYEE + CHILD, the minimum set of data elements that functionally determines every data element in either relation. (In R, the FDs:

EMPLOYEE+CHILD -----> ADDRESS

EMPLOYEE+CHILD ----->AGE

follow from the FDs:

EMPLOYEE -----> ADDRESS

EMPLOYEE -----> AGE

by inference rule FD2.) R2 thus is all-key.

If X is a key of a relation R, then each data element of X is called a *prime* data element of R. Any other data element of R is *non-prime* in R. (A data element can be prime in one relation and non-prime in another.) A subset Y of the data elements of R is called a *key set* of data elements of R (often referred to as a *superkey* of R) if Y either is a key of R or contains at least one key as a proper subset (in the latter case, we refer to Y as a *proper superkey* of R). Any other subset of R's data elements is *non-key* in R. A non-key set of data elements of a relation R can contain data elements that are prime in R (if R has a composite key, each data element of the key is prime in R but forms a non-key data element set in R).

A non-key set of data elements in a relation R is a *foreign key* in R if it is a key of another relation, which is called the *target relation*. In the above example, the data element EMPLOYEE is a foreign key in R2 because it is non-key in R2 and is a key of R1.

If a key of a relation R2 is multiply dependent on a non-key set of data elements in relation R1 (where R1 and R2 can be the same relation), the non-key set is a *secondary key* to R2. R1 is the *source relation* and R2 is the *target relation*. A foreign key in one relation can be a secondary key in the same or another relation. Consider, for example, these relations:

R1 (EMPLOYEE, ADDRESS, AGE, DEPARTMENT) with key EMPLOYEE and associated MVD
DEPARTMENT ----->> EMPLOYEE

R2 (EMPLOYEE, CHILD) with key EMPLOYEE+CHILD

R3 (DEPARTMENT, MANAGER, BUILDING) with key DEPARTMENT

DEPARTMENT is a foreign key in R1 with R3 as the target relation, EMPLOYEE is a foreign key in R2 with R1 as the target, and DEPARTMENT is a secondary key to R1 where R1 is also the source relation.

Note:

DEPARTMENT is both a secondary key and a foreign key in R1.

Specification of keys in a relational approach can serve several useful purposes. First of all keys are unique identifiers of rows in relation instances and thus provide access to the content of a database. This is of particular importance at the implementation level. Secondly they provide a convenient method of representing FDs. In the synthesis approach to design, all input FDs (whether or not eliminated during normalization) are represented in the generated relations through the keys, directly or indirectly. Formally, the FD:

$X \rightarrow A$

is *embodied* in a relation R if X is a superkey of R and A is a data element of R. In the Bernstein synthesis process, for every FD:

$X \rightarrow A$

remaining after normalization, X becomes a key of some relation and not a proper superkey of any (proper superkeys are averted during normalization by elimination of extraneous data elements from the left-hand side of FDs). Thus, a relation scheme (see ["Relations and Dependencies" on page 223](#)) for a relation R can be formulated as a set T of data elements for R, a list of one or more subsets of T specified as keys of R, and a list of any MVDs associated with R, each defined over T. The set of FDs embodied in a relational schema contains every FD embodied in any relation of the schema. The set of FDs *represented* in a schema is the closure of the set of FDs embodied in it (that is, all the FDs derivable from the embodied FDs using the rules of inference given in ["Rules of Inference for Deriving Additional FDs" on page 225](#)). It can be proved that the set of FDs embodied in a schema generated by the Bernstein synthesis procedure is a covering for the set F of FDs input to the procedure. Therefore, such a schema represents the closure F^+ of all the FDs input. The schema then is said to have the *faithful closure* property (see [reference 4 on page 257](#)). Since all the FDs of F (being a subset of F^+) are represented in a schema with the faithful closure property, the schema is also said to have the *faithful representation* property.

Finally, keys are useful because they define many of the *associations* between generated relations, that is, the *hierarchical* and *foreign key* associations. *Secondary key* associations are in part defined by keys. (See ["Glossary" on page 245](#) for definitions and discussion, with examples, of the various associations.)

The Normal Forms Defined

Some of the benefits of a normalized schema are discussed and illustrated in ["Generating a Third Normal Form Relational Schema" on page 132](#). They are presented in terms of structural stability achieved and reduction of data redundancy in implementation, leading to reduction in maintenance costs and storage requirements and to elimination of many problems associated with adding, deleting, and updating data values. Normalization is the application of those procedures which lead to generation of schemas in any of the various normal forms. DesignManager produces schemas which are in third normal form (3NF).

To introduce the normal forms, some additional definitions are required and are described here.

A data element A is *fully functionally dependent* on a set X of data elements (not including A) if A is functionally dependent on X but not on any subset of X .

Let X be a subset of the set of data elements of a relation R , and let A be any data element of R . Then A is *transitively dependent* on X (in R) if there exists a set Y of data elements of R (not containing A) such that the FDs:

$X \text{ -----} > Y$

$Y \text{ -----} > A$

are valid in R (but not $Y \text{ -----} > X$).

An FD $X \text{ -----} > Y$ is *trivial* (it is valid in any relation containing X) if $Y \subset X$. Otherwise it is *non-trivial*.

If R is a relation and X and Y are subsets of R 's data elements, then the MVD:

$X \text{ -----} >> Y$

is *trivial in R* (it must be valid in R) if the set of all of R 's data elements is precisely the union of X and Y . Otherwise (that is, if R has data elements that are neither in X nor in Y), the MVD is *nontrivial in R* .

For example, this MVD is nontrivial in $R(A,B,C)$:

$A \text{ -----} >> B$

Whereas this MVD is trivial in R :

$A+B \text{ -----} >> C$

Other than first normal form (which restricts the domain of every data element of a relation to simple values and has nothing to do with dependencies), the normal forms of interest for the synthesis and/or decomposition approaches to schema design are second, third, and Boyce-Codd normal forms, see [reference 6 on page 257](#) and [reference 10 on page 258](#)) and fourth normal form (see [reference 9 on page 258](#)), defined and discussed in these paragraphs.

Second Normal Form (2NF)

A relation R is in *2NF* if each of its non-prime data elements is fully functionally dependent on each of its keys. Thus an all-key relation is automatically in 2NF. A schema is in 2NF if each of its relations is in 2NF.

Given, for example, the FDs:

```
SUPPLIER -----> ADDRESS
SUPPLIER+PART ----> PRICE
```

the relation R(SUPPLIER,PART,ADDRESS,PRICE) is not in 2NF. The only possible key to R is SUPPLIER+PART; however, ADDRESS, being functionally dependent on SUPPLIER, is not fully functionally dependent on the key. The synthesis procedure instead would produce the two relations:

```
R1 (SUPPLIER, ADDRESS) with key SUPPLIER
R2 (SUPPLIER, PART, PRICE) with key SUPPLIER+PRICE
```

The presence of MVDs can also affect whether or not a relation is in 2NF. This is illustrated in ["Relations and Dependencies" on page 223](#) and ["Keys and Representation of FDs in a Schema" on page 227](#) by the relation R(EMPLOYEE,ADDRESS,AGE,CHILD), which is not in 2NF because the non-prime data elements ADDRESS and AGE are functionally dependent on just one of the components (EMPLOYEE) of the key EMPLOYEE + CHILD. The separation required to obtain a 2NF schema is achieved in the synthesis process through generation of relation R1(EMPLOYEE,ADDRESS,AGE) from the FDs:

```
EMPLOYEE -----> ADDRESS
EMPLOYEE -----> AGE
```

and R2(EMPLOYEE,CHILD) from the MVD:

```
EMPLOYEE ---->> CHILD
```

Third Normal Form (3NF)

A relation R is in *3NF* if none of its non-prime data elements is transitively dependent on any of its keys. Thus an all-key relation is automatically in 3NF. A schema is in 3NF if each of its relations is in 3NF.

In the example of ["An Example Illustrating Normalization Benefits" on page 137](#), the relation R(SUPPLIER,CITY,CODE) is not in 3NF because the non-prime data element, CODE, is transitively dependent on the key SUPPLIER via the FDs:

```
SUPPLIER -----> CITY
CITY -----> CODE
```

The synthesis procedure instead would eliminate the FD:

```
SUPPLIER -----> CODE
```

as transitively redundant and produce the 3NF relations:

R1 (SUPPLIER, CITY) with key SUPPLIER
R2 (CITY, CODE) with key CITY

Remark 1. Every 3NF relation is automatically in 2NF. For, if a non-prime data element A is not fully functionally dependent on a (composite) key, it must be transitively dependent on that key via a subset of the key. In the SUPPLIER/PART illustration of 2NF, for example, the relation R(SUPPLIER,PART,ADDRESS,PRICE) is not in 3NF because the FDs:

SUPPLIER+PART -----> PRICE
SUPPLIER -----> ADDRESS

both hold in R, causing the non-prime data element ADDRESS to be transitively dependent on the key SUPPLIER + PART via the data element SUPPLIER.

The definition of 3NF (and of 2NF) is concerned with characteristics of a relation's non-prime data elements only. While 3NF is sufficient to remove many of the updating problems for relational schemas, some of the problems may still be present for prime data elements in FD-relations. As a consequence, an even stronger normal form, Boyce-Codd normal form (BCNF) was developed (see [reference 10 on page 258](#)) which removes this distinction between prime and non-prime data elements. In many instances, BCNF provides a better schema and still represents all the input FDs. However, there are examples for which it is impossible to obtain a BCNF schema with the faithful representation property. Furthermore, even when such a schema does exist, there is no efficient algorithm known for producing it (see [reference 1 on page 257](#)). Following the definition of BCNF, an example is presented (from [reference 1 on page 257](#)) for which such a schema does not exist.

Boyce-Codd Normal Form (BCNF)

A relation R is in *BCNF* if, for every non-trivial FD

X -----> A

valid in R, X must be a superkey of R. That is, if X -----> A holds for any data element A of R (whether prime or non-prime), then it must hold for all data elements of R. An all-key relation R is *not* automatically in BCNF; a data element of R may be functionally dependent on a subset X of the others, yet X cannot be a superkey since all of R's data elements comprise the only possible key (by definition of an all-key relation). A schema is in BCNF if all of its relations are in BCNF.

Remark 2. Even Fagin (in [reference 7 on page 257](#) and [reference 9 on page 258](#)) falls into the trap of claiming that an illustrative relation, T(EMPLOYEE,CHILD,SALARY,YEAR), is in BCNF because it is all-key. If relation T is in BCNF, it is not because it is all-key, which also happens to be true. It would be in BCNF because no non-trivial FD has been specified which is valid in T. (Actually, the non-trivial FD:

EMPLOYEE+SALARY -----> YEAR

probably is, on semantic grounds, valid in T, in which case T would not be in BCNF because the FD, EMPLOYEE+SALARY -----> CHILD, is not valid in T.)

Remark 3. Zaniolo (in [reference 8 on page 258](#)) proves that the following alternative formulation of 3NF is equivalent to the definition given above; it is analogous to the definition of BCNF and so provides a basis for comparison:

Third Normal Form. A relation R is in 3NF if, for every nontrivial F: X \rightarrow A, valid in R, either:

- (a) X is a superkey of R.
- (b) A is a prime data element of R.

The definitions are identical except for condition (b). For BCNF, condition (a) must be true whether or not A is a prime data element, whereas, for 3NF, it need not be true when A is prime. BCNF is stronger and implies 3NF; that is, if R is in BCNF, then it must be in 3NF.

This example ([reference 1 on page 257](#)) illustrates a relation which is in 3NF but not in BCNF, nor can it be achieved by an efficient algorithm. Let F be the set of FDs shown here:

STREET-ADDRESS+CITY \rightarrow POSTAL-CODE
 POSTAL-CODE \rightarrow CITY

If input to the synthesis algorithm (see ["The Extended Synthesis Procedure for Schema Design" on page 238](#)), these relations would generate:

R1 (STREET-ADDRESS, CITY, POSTAL-CODE) with key STREET-ADDRESS+CITY
 R2 (POSTAL-CODE, CITY) with key POSTAL-CODE

Although both relations are in 3NF, relation R1 is not in BCNF since POSTAL-CODE functionally determines one, but not all, of R1's data elements. Furthermore, it can easily be proved (see [reference 1 on page 257](#)) that there is no BCNF schema that can represent the above FDs. To achieve BCNF, R1 must be decomposed yielding, for example, the relations:

R1A (POSTAL-CODE, STREET-ADDRESS) which is all-key
 R2 (POSTAL-CODE, CITY) as above

(This is a legitimate decomposition in the sense of Codd or Fagin because the natural join of R1A and R2, in all instances, is precisely R1 and thus the data content of R1 is preserved by the decomposition.) It is evident that the first FD given above is not represented by this schema. A further objection to the schema is that the two relations cannot be updated independently, thus defeating a major goal of the separation. If, say, a new value of POSTAL-CODE is assigned to one or more combinations of CITY and STREET-ADDRESS, both relations must be updated.

A natural extension of BCNF for the case of multivalued dependencies (MVDs) is fourth normal form (4NF), but in general this is true only for the case of Fagin-MVDs. Indeed, Fagin-MVDs provide a necessary and sufficient condition for a relation to be decomposable into two of its projections without loss of information, that is, in such a way as to preserve the data content of all its instances. As with BCNF, there are many instances in which 4NF provides a better schema and still represents all the input dependencies. However, since a 4NF relation must be in BCNF (see below), the same sort of disadvantages can arise. In particular, there are examples for which it is impossible to obtain a 4NF schema with the faithful representation property. Even when such a schema exists, there is no efficient algorithm known for producing it. The definition of 4NF is given below, following the definition of a Fagin-MVD.

A *Fagin-MVD* $X \twoheadrightarrow Y$, where X and Y are sets of data elements, is a DSR-MVD (see ["Relations and Dependencies" on page 223](#)) that satisfies these further conditions (any FD valid in a relation also satisfies these conditions):

- It is formulated in the context of a relation $R(X,Y,Z)$ in which it holds, where Z is the remaining set of R 's data elements (if there are no other data elements, then Z is the null set).
- If it is non-trivial in R (that is, Z is not the null set), then, for every instance of R and its two projections $R_1(X,Y)$ and $R_2(X,Z)$, the natural join of the R_1 -instance and the R_2 -instance is just the R -instance. Symbolically, this is expressed as $R = R_1 * R_2$ and is termed a *lossless join* to indicate that the decomposition to R_1 and R_2 preserves the data content of every instance of R . Equivalently, it can be said that Y and Z are *independent* (or *orthogonal*) in R with respect to X . That is, in any table of values constituting an instance of R , if to a given X -value there correspond m Y -values and n Z -values, then there must be mn rows of the table which contain the given X -value, corresponding to every distinct combination of Y - and Z -values.

A schema derived from a given set of relations by lossless join decomposition is said to have the *lossless join property* (with respect to the given set).

Remark 4. It is evident from the definition that, if $X \twoheadrightarrow Y$ is a non-trivial Fagin-MVD in $R(X,Y,Z)$, then so is $X \twoheadrightarrow Z$.

Lossless join and the lossless join property are defined above in terms of the decomposition approach to schema design. They are concerned with the decomposition of a relation into two of its projections and as such are not applicable to a schema obtained by synthesizing relations from dependencies. In the following definition, the concept of lossless join is generalized to any number of projections of a given relation and attention is focused on the case of schemas obtained (by any method whatever) from a schema consisting of a single relation only. Given a (universal) relation $U(T)$, where T is the set of all the data elements of interest (that is, in the database), and given a schema:

$$S = \{R_1(T_1), R_2(T_2), \dots, R_n(T_n)\}$$

where the T_i are sets of data elements, not necessarily disjoint, the schema S has the *lossless join property* (see [reference 2 on page 257](#) and [reference 4 on page 257](#)) if these conditions hold:

- The set union of the T_i is exactly T .
- Given any instance of U and its projection on each of the sets T_i , the natural join of the projections is exactly the given U -instance. (Although the join is a binary operator, it is also associative and commutative and can be applied unambiguously to more than two relations without specifying the order of operation; see [reference 13 on page 258](#) and [reference 12 on page 258](#).)

This definition can be applied to a schema generated by a synthesis procedure as well as a decomposition procedure, or any combination of the two, provided that Bernstein's *universal relation assumption* is made (see [reference 1 on page 257](#) and [reference 2 on page 257](#)). That is, given a relational database:

$$R_i(T_i), \quad i = 1, 2, \dots, n$$

where T is the union of the T_i ; there is a single universal relation $U(T)$ such that, for every instance of U , the corresponding instance of R_i is just the projection of the U -instance on T_i , for $i = 1, 2, \dots, n$.

Not every 3NF schema obtained by the synthesis approach described in "[The Extended Synthesis Procedure for Schema Design](#)" on page 238 satisfies the lossless join property. Indeed, most probably do not. However, where only FDs are specified, Biskup ([reference 4 on page 257](#)) shows that a schema synthesized via the Bernstein process can be transformed to one with the lossless join property without violating the faithful closure property, and thus the faithful representation property), which already holds for the synthesized schema. It can further be conjectured that a schema synthesized by the DesignManager design procedure (whether or not MVDs and MVD-relations are present) can similarly be transformed if attention is restricted only to the FD-relations.

Remark 5. There are examples which show that a relation can be the join of three of its projections but not the join of any two (see [reference 4 on page 257](#), [reference 13 on page 258](#), [reference 14 on page 258](#), and [reference 12 on page 258](#)). In particular, proof is indicated in [reference 14](#) that this is true for all n greater than 2. That is, for each n there is a relation R with n projections having the lossless join property (with respect to R) but with no set of fewer than n projections having the same property. This phenomenon is the basis of the *join dependency* and fifth normal form (see [reference 12 on page 258](#) and [reference 13 on page 258](#)), also called *projection-join normal form* by Fagin in [reference 13 on page 258](#).

As an example (see [reference 9 on page 258](#)) illustrating Fagin-MVDs, consider this relation which specifies the children and salary history of each employee:

$$R(\text{EMPLOYEE}, \text{CHILD}, \text{PAST-SALARY}, \text{SALARY-DATE})$$

Intuitively, it is obvious that the two are independent for each employee and thus that these MVDs are Fagin-MVDs:

```
EMPLOYEE ---->> CHILD
EMPLOYEE ---->> PAST-SALARY+SALARY-DATE
```

If, for example, employee Adams has two children, Paul and Jane, and has received the three salaries, 20K, 23K, and 25K, starting respectively in the years 1977, 1979, and 1982, then the Adams portion of the table for the current R-instance would require six rows, as shown in [Figure 12](#).

Figure 12. Partial Instance of Fagin-MVD

| R | | | |
|----------|-------|-------------|-------------|
| EMPLOYEE | CHILD | PAST-SALARY | SALARY-DATE |
| Adams | Paul | 20K | 1977 |
| Adams | Paul | 23K | 1979 |
| Adams | Paul | 25K | 1982 |
| Adams | Jane | 20K | 1977 |
| Adams | Jane | 23K | 1979 |
| Adams | Jane | 25K | 1982 |

Consider the projections of this table on R1(EMPLOYEE,CHILD) and R2(EMPLOYEE,PAST-SALARY,SALARY-DATE), as shown in [Figure 13](#). The join of the two projections results again in the R-instance shown in [Figure 12](#) indicating that the MVDs given above are indeed Fagin-MVDs. On the other hand, the values of PAST-SALARY are not independent from the values of SALARY-DATE, that is, it would be impossible to reconstruct the R2 table from the join of the projections shown in [Figure 14 on page 237](#).

Figure 13. Projections of the R-instance shown in Figure 12 which illustrate the definition of Fagin-MVD

| R1 | |
|----------|-------|
| EMPLOYEE | CHILD |
| Adams | Paul |
| Adams | Jane |

| R2 | | |
|----------|-------------|-------------|
| EMPLOYEE | PAST-SALARY | SALARY-DATE |
| Adams | 20K | 1977 |
| Adams | 23K | 1979 |
| Adams | 25K | 1982 |

Figure 14. Projections of the R-instance shown in figure 12 which do not satisfy the definition of Fagin-MVD

| R21 | |
|----------|-------------|
| EMPLOYEE | PAST-SALARY |
| Adams | 20K |
| Adams | 23K |
| Adams | 25K |

| R22 | |
|----------|-------------|
| EMPLOYEE | SALARY-DATE |
| Adams | 1977 |
| Adams | 1979 |
| Adams | 1982 |

Indeed the join of R21 and R22 would consist of 9 rows, representing every combination of PAST-SALARY with SALARY-DATE, rather than three rows (which would be required for a lossless join). As a consequence, although these DSR-MVDs can be specified by the user if desired, neither of them satisfy the definition of Fagin-MVD:

```
EMPLOYEE ---->> PAST-SALARY
EMPLOYEE ---->> SALARY-DATE
```

As seen from the following definition, these DSR-MVDs cannot be used as the basis for 4NF decomposition, in contrast with the Fagin-MVDs given above which can.

Fourth Normal Form (4NF)

A relation R is in *4NF* if it is in BCNF and no non-trivial Fagin-MVD (other than an FD) holds in R. A schema is in 4NF if each of its relations is in 4NF.

The example given above with [Figure 12 on page 236](#) through [Figure 14](#) above illustrates the 4NF decomposition process. Given this relation and the Fagin-MVDs:

```
R (EMPLOYEE, CHILD, PAST-SALARY, SALARY-DATE)
```

This resulting schema is in 4NF:

```
R1 (EMPLOYEE, CHILD)
R2 (EMPLOYEE, PAST-SALARY, SALARY-DATE)
```

Furthermore, it represents the input dependencies.

Remark 6. The DesignManager synthesis procedure would not generate the non-4NF relation R from the Fagin-MVDs given in the previous example. Indeed, it would automatically produce the 4NF relations R1 and R2. This sort of automatic extendability of the normalization procedure is further evidence of the robustness of the Bernstein synthesis procedure (see the footnote to [reference 8](#) in the ["Bibliography" on page 257](#)).

Although the above example illustrates the benefits of 4NF decomposition (identical data content is depicted much more efficiently in [Figure 14 on page 237](#) than in [Figure 12 on page 236](#)), as with BCNF, there are examples for which 4NF decomposition is not desirable. Indeed, the POSTAL-CODE/CITY/STREET-ADDRESS illustration given above is just such an example since BCNF is one of the conditions required for 4NF. In particular, consider this FD:

POSTAL-CODE -----> CITY

Use of the FD to split R1 (STREET-ADDRESS,CITY,POSTAL-CODE) into R1A(POSTAL-CODE,STREET-ADDRESS) and R2(POSTAL-CODE, CITY), as an example of BCNF decomposition, is also an example of 4NF decomposition. The undesirable effects of the decomposition are indicated above.

The Extended Synthesis Procedure for Schema Design

Following is a description of the extended synthesis procedure for generating a 3NF relational schema from a given set of dependencies. Input to the design procedure is a set F of FDs and a set M of MVDs (the workbench content after a MERGE command), defined over a set T of data elements. The procedure described is a modification of the Bernstein synthesis procedure for FDs (see [reference 1 on page 257](#) and [reference 3 on page 257](#)) extended to handle MVDs.

1. Elimination of Extraneous Data Elements

Each FD, f , of F with two or more data elements on its left-hand side is examined to see if any of these data elements can be eliminated as extraneous. In particular, the FD which would result from such elimination is tested to see if it is derivable from F via the FD inference rules given in ["Rules of Inference for Deriving Additional FDs" on page 225](#). (Examples of this step are given in ["The DesignManager Design Procedure" on page 138](#).) Thus, if f is the FD:

X -----> A

and B is a data element in X, then B is *extraneous in f* (relative to F) if this FD is in F^+ :

X - {B} -----> A

If an FD has no extraneous data elements (on its left-hand side), it is said to be *reduced*. Let G be the set of reduced FDs obtained from F (including FDs of F already in reduced form). Then $G^+ = F^+$, so that G is a covering of F . To prove this, it only needs to be shown that each FD:

$X \text{ -----} \rightarrow A$

of F is in G^+ (it is already known that each FD of G is in F^+ by definition of G). For this, it is sufficient to show that a reduced FD:

$X - Y \text{ -----} \rightarrow A$

of G implies the corresponding unreduced FD:

$X \text{ -----} \rightarrow A$

of F from which it was obtained, where Y is the set of all data elements eliminated as extraneous from X , the left-hand side of the unreduced FD. This follows immediately by applying inference rule FD2 (augmentation) to the reduced FD, taking W as the set Y and Z as the null set.

Remark 1. The set G is not necessarily unique. The data elements appearing in the left-hand side of a reduced FD may depend on the order in which data elements were selected for testing in the corresponding unreduced FD of F . Furthermore, there may be redundant FDs in G . Thus, although G is a covering for F , it will not in general be a nonredundant covering. However, it does ensure that, in [step 6 on page 243](#), the schema generated will be in 2NF, or, what is essentially the same thing, that the left-hand side of every FD of H (the set of FDs remaining after [step 2 on page 240](#)) becomes a key of some relation generated and not a *proper* superkey of any.

2. Removal of Redundant FDs

Each FD, f , of G (the set of reduced FDs obtained in Step 1) is tested to see if it is redundant in G , that is, to see if f is in $(G - \{f\})^+$. Examples are given in ["Step 2: Removal of Redundant Dependencies" on page 140](#); in the second example, this FD removed is derivable from the others by inference rule FD3 (pseudotransitivity).

$A+D \text{ -----} > C$

When all redundant FDs have been removed from G , the resulting subset H of G is a nonredundant covering of G and thus of F , that is, $H^+ = G^+ = F^+$.

Remark 2. The composition of H after this step is not necessarily unique, either in content or number of FDs. It may depend on the order in which FDs are selected for testing. However, because it is nonredundant, it does guarantee generation of a 3NF schema in [step 6 on page 243](#).

After all redundant FDs are removed from G , each MVD, m , in M is tested in turn (as if it were an FD) to see if it is derivable from the FDs of H . In particular, if m is the MVD:

$X \text{ ----} >> Y$

then, letting f be the FD $X \text{ ----} > Y$, a test is made to see if f is in H^+ , that is, to see if f is redundant in the set $H \cup \{f\}$. If so, both f and m are in the set $F^+ \cup M$. This represents an inconsistency in the input set $F \cup M$ of dependencies since f and m cannot both be valid if m is a *proper* MVD. In the DesignManager design procedure, the problem is provisionally resolved by removing m from M (that is, as an FD, f , redundant in the set $H \cup \{f\}$), setting an *inconsistency* flag internally, and providing a warning message to indicate that the current design was produced from inconsistent input. For each *inconsistent* MVD, m , removed from M , set M to $M - \{m\}$.

3. Partitioning of H into FD-Groups

As a preliminary to forming relations ([step 6 on page 243](#)), this step 3 through [step 5 on page 242](#) are concerned with the partitioning of $H \cup M$ into groups of dependencies, based on the potential keys of the relations to be formed. A *potential key* of a group of dependencies is a set of data elements appearing in one or more dependencies of the group which functionally determines all the data elements in the dependencies of the group. A potential key of a group will be designated as a key of a relation generated in [step 6 on page 243](#). In the partition, two kinds of groups are formed, *FD-groups* and *MVD-groups*. Each FD-group contains one or more FDs and perhaps some MVDs. Each MVD-group contains one or more MVDs. A comprehensive example is given in [step 4 on page 241](#) and [step 5 on page 242](#). In this step, the nonredundant set H of FDs is partitioned into FD-groups, each with a single potential key and containing all FDs having the same left-hand side. Formally, in this step, for each distinct left-hand side X_i , $i = 1$ to (say) p , appearing in any FD of H , the FD-group H_i formed is the set of all FDs whose left-hand side is X_i (the potential key of H_i), that is:

$H_i = \{f \mid X_i \text{ is the left-hand side of } f\}, i = 1 \text{ to } p$

4. Partitioning of M into MVD-Groups

FD-groups form from the set F of FDs in [step 3 on page 240](#). Similarly, the set M of MVDs is partitioned into groups of MVDs in this step and [step 5 on page 242](#). In this step, MVD-groups are formed each containing a single MVD. In [step 5 on page 242](#), MVD-groups having the same set of data elements (that is, the same potential key) are merged. However, it is not necessarily true that every MVD is used to form an MVD-group; under certain conditions (see case 1 below), some MVDs will be placed in existing FD-groups. The intent is to generate 3NF relations (in [step 6 on page 243](#)) from the groups such that no two relations (whether FD-relations or MVD-relations) have the same key. It is also intended that every MVD-relation generated be all-key. In [step 4](#), two cases arise:

Case 1. If the set of data elements comprising an MVD both contains (as a subset) or is identical to the potential key of an FD-group and is itself a subset of the FD-group's set of data elements or identical to it, then the MVD is placed in the FD-group rather than being used to form an MVD-group.

Case 2. An MVD which does not satisfy the conditions of Case 1 is used to form an MVD-group whose potential key is the MVD's entire set of data elements.

Remark 3. The MVD-groups formed in [step 4](#) and [step 5 on page 242](#) are used, as indicated above, to generate MVD-relations which are all-key. If MVDs satisfying the conditions of case 1 were not placed in FD-groups but instead were used to form MVD-groups, they would in [step 6 on page 243](#), give rise to MVD-relations which would not necessarily be all-key. Moreover, the potential key of each such MVD-group would be identical to the potential key of one of the FD-groups. In both respects, this would not conform to the intent stated above.

Remark 4. It is possible for a given MVD to satisfy the conditions of case 1 for more than one FD-group. In that event, the MVD is placed in the first FD-group examined. However, the possibility of this sort of order-dependence is not crucial. What is important is to produce 3NF relations with distinct keys where MVD-relations are all-key.

Formally, [step 4 on page 241](#) is performed as follows: For each FD-group, H_i , $i = 1$ to p , let X_i be the potential key of H_i and let Z_i be the set of data elements appearing in the FDs of H_i . Given an MVD, m , let Y be the set of data elements appearing in it. Consider an FD-group H_i . If $X_i \subseteq Y$ and $Y \subseteq Z_i$, then m is placed in H_i (case 1) and the procedure begins again for the next MVD. If the MVD m does not meet these conditions for any value of i , the MVD-group $\{m\}$ is formed with Y as its potential key, and the procedure is repeated for the next MVD.

Remark 5. There may be several MVD-groups composed of the same set of data elements and thus having the same potential key: these are merged in [step 5 on page 242](#). More than one MVD can be placed in the same FD-group; these need not be composed of the same set of data elements.

To illustrate the procedure, assume that these FD-groups have been formed in [step 3 on page 240](#):

H1 : A+B -----> C, A+B -----> D with A+B as potential key
H2 : D -----> B, D -----> E with D as potential key.

Suppose these MVDs are also given, labeled, respectively, m1 to m8:

A ---->> B, B ---->> A, B+C ---->> A, F ---->> A+B,
C ---->> D+A, D ---->> C+A, B ---->> D, D ---->> G

Using the notation given above, these are the FD-groups H1 and H2:

Z1={A,B, C, D}, X1 = A+B (that is, {A, B})

Z2={D, B, E}, X2 = D (that is, {D})

These are the respective MVDs:

Y1 = {A, B}, Y2 = {B, A}, Y3 = {B, C, A}, Y4 = {F, A, B},
Y5 = {C, D, A}, Y6 = {D, C, A}, Y7 = {B, D}, Y8 = {D, G}

Note: _____

Y1 = Y2 and Y5 = Y6.

Treating each MVD, mj, in turn, j = 1 to 8:

- j = 1,2: Y1 = Y2 = X1;
m1 and m2 are placed in H1.
- j = 3: Y3 contains X1 and is contained in Z1; m3 is placed in H1.
- j = 4: Y4 contains X1 but is not contained in Z1; Y4 does not contain X2;
m4 is used to form an MVD group {m4} with Y4 as potential key.
- j = 5,6: Y5 and Y6, although contained in Z1, do not contain X1;
Y5 and Y6 do not contain X2;
m5 and m6 are used to form the respective MVD-groups, {m5} and {m6}, both having Y5 = Y6 as potential key.
- j = 7: Y7, although contained in Z1, does not contain X1; Y7 contains X2
and is contained in Z2;
m7 is placed in H2.
- j = 8: Y8 does not contain X1;
Y8 contains X2 but is not contained in Z2;
m8 is used to form the MVD-group {m8} with Y8 as potential key.

5. Merging MVD-Groups with Identical Potential Keys

As indicated in [remark 5 on page 241](#), there may be instances in which several MVD-groups are composed of the same set of data elements and thus have the same potential key. In accordance with the goal of producing relations with distinct keys, all MVD-groups having a potential key in common are merged to form a single MVD-group with the same potential key and containing each of the MVDs belonging to the groups merged. In the example of [step 4 on page 241](#), MVD-groups {m5} and {m6} are merged in this step to form a single MVD-group containing both MVDs and having the common potential key {C,D,A}. At the end of this step, the set HuM has been partitioned into p FD-groups Hi, i = 1 to p, and (say) q MVD-groups Mj, j = 1 to q, with p+q distinct (relabelled) potential keys Xi, i = 1 to p+ q, where some of the MVDs of M may be contained in FD-groups.

6. Generation of 3NF Relations

For each group of dependencies present at the end of [step 5 on page 242](#), a relation is constructed consisting of all the data elements appearing in any of the dependencies of the group. The potential key of the group is identified as the key of the relation. A relation formed from an FD-group is called an FD-relation and will have as its key the left-hand side common to the FDs of the group ([step 1 on page 238](#) guarantees that the common left-hand side is a key and not merely a proper superkey). Each FD and MVD in the FD-group is associated with the generated FD-relation, the FDs being embodied via the key. A relation formed from an MVD-group is called an MVD-relation and will be all-key. Each MVD of the MVD-group is associated with the MVD-relation. The separation from FD-relations of those MVDs which are associated with MVD-relations is necessary to ensure that the FD-relations are in 3NF (or even 2NF; see ["The Normal Forms Defined" on page 230](#)). An MVD associated with an FD-relation, on the other hand, causes no problem in this regard because its set of data elements is functionally dependent precisely on the left-hand side common to the embodied FDs. The set of constructed relations constitutes a schema for the input set FuM of FDs and MVDs.

DSR-MVD Versus Fagin-MVD

It was noted in ["Relations and Dependencies" on page 223](#), that the DesignManager definition of MVD is less restrictive than that given by Fagin. This is obvious from a comparison of the definitions. Fagin's formulation requires the existence of a relation R in which a Fagin-MVD must hold. This is not required of a DSR-MVD. When a Fagin MVD:

X ---->> Y

is non-trivial in a relation R(X,Y,Z), it is further required that Y and the remaining set Z of R's data elements be orthogonal in R with respect to X. This condition need not be met for a DSR-MVD which is non-trivial in a relation. The only purpose in adding these restrictive conditions is to provide a basis for 4NF decomposition, which, for a relation R, is permitted only on the basis of a valid non-trivial (in R) Fagin-MVD, thus ensuring preservation of data content in the decomposition (see ["The Normal Forms Defined" on page 230](#)). That is, any instance of R must be the join of the projections resulting from its decomposition.

By divorcing the definition of an MVD from the process of 4NF decomposition, DesignManager provides these advantages:

- The user can formulate MVDs in a more *natural* fashion, as seen fit, without being forced to consider the orthogonality of sets of data elements within a containing relation. Indeed, at input time, the user doesn't know what the containing relations will be; this is especially true in a large database. A user should not be prevented, for example, from specifying the MVD:

SUPPLIER-NO ---->> PART-NO

if required for the application, no matter what other data elements appear in the database and no matter what other dependencies may be specified in the various userviews, such as:

SUPPLIER-NO+PART-NO -----> PRICE, or

SUPPLIER-NO ----->> PART-NO+PRICE

- If a procedure for decomposing MVD-relations is introduced (say manually) by the user after DesignManager has synthesized 3NF relations (see "[The Extended Synthesis Procedure for Schema Design](#)" on page 238), the use of DSR-MVDs at input time would enable the user to defer a decision as to which are Fagin-MVDs until the time that manageable relations have been produced. (Any such decomposition must be based on non-trivial Fagin-MVDs to ensure preservation of data content; recall that any FD is a Fagin-MVD.) The point is, that it is much easier at this point to make an accurate decision than it is at input time when no proper relation is available.

Consider this example of FDs from an input userview:

EMP-NO -----> DEPT-NO
EMP-NO -----> EMP-NAME
EMP-NO -----> EMP-ADDR
EMP-NO -----> AGE

It would be perfectly legitimate for the user to add this DSR-MVD:

DEPT-NO ----->> EMP-NO

Whereas the only valid Fagin-MVD would be this:

DEPT-NO ----->> EMP-NO+EMP-NAME+EMP-ADDR+AGE

Moreover, if a second user had input the FD:

EMP-NO -----> POSITION

(perhaps without the knowledge of the first user), then the second MVD above still would not be a Fagin-MVD. By permitting DSR-MVDs, such as the first one above, no such difficult decision is required at input time.

This glossary provides the terms specific to DesignManager and its use.

Absolute Dependency Number

During processing of the MERGE command (see ["MERGE" on page 79](#)), each dependency entered into the workbench design area is assigned two numbers:

- An absolute dependency number indicating its position in the workbench design area
- A relative dependency number indicating the order in which it is added to the workbench design area from the USERVIEW member being processed.

These numbers are used in output reports produced by various design mode commands.

All-key Relation

See ["Key" on page 250](#).

Alternative Key

See ["Key" on page 250](#).

Association between Relations

A unidirectional or bidirectional relationship between two relations indicating one of these:

- A dependency exists between the keys of the relations (see ["Foreign Key Association" on page 248](#) and ["Multivalued Association" on page 250](#)).
- The key of one relation is contained in the key of the other (see ["Hierarchical Association" on page 249](#)).
- An MVD exists from a non-key set of data elements of one relation to the key of the other (see ["Secondary Key Association" on page 254](#)).

Attribute

Another term for *data element* in DesignManager, data elements that characterize an entity are called attributes.

Bijection

See ["Equivalence of Data Elements" on page 247](#).

Boyce-Codd Normal Form (BCNF)

A relation is in BCNF if, for every non-trivial FD $A \twoheadrightarrow B$ which holds in R, each data element of R is functionally dependent on A. If no non-trivial FD holds in R (that is, R is all-key), then R is automatically in BCNF. If a relation is in BCNF, then it must be in 3NF.

Composite Key

A key (of a relation) composed of two or more data elements (also referred to as a concatenated key).

Composite View

Before the DESIGN command is entered to generate a 3NF schema from a collection of userviews, the userviews are merged into the workbench design area (via one or more MERGE commands) to form a composite view, consisting of:

- All the FDs and MVDs entered in the USERVIEWS, with all duplications and inconsistencies (see [remark 6 on page 81](#) of the ["MERGE" on page 79](#) command) removed.
- Any implied FDs automatically generated and added by DesignManager.

Concatenated Key

See ["Composite Key" on page 246](#).

Data Element

The basic unit of a data base model; that is, the smallest unit to which data values can be assigned (also referred to in the literature as an *attribute*). Data elements are entered and stored in the modeling dictionary as ITEM or GROUP members.

Data Element Number

During processing of the MERGE command (see ["MERGE" on page 79](#)), each data element, when added to the workbench design area as part of the left-hand side or right-hand side of a dependency is assigned a unique data element number. The first data element added is assigned the number 1, the second the number 2, etc.

Data-View

Generic category indicating either a userview or an entity.

Data-view Number

During processing of the MERGE command (see ["MERGE" on page 79](#)), each data-view added to the workbench design area is assigned a unique number. The first dataview added is assigned the number 1, the second the number 2, etc.

Dependency

See ["Functional Dependency \(FD\)" on page 248](#) and/or ["Multivalued Dependency \(MVD\)" on page 251](#).

Derivability (of FDs)

An FD is derivable from a given set of FDs if it can be obtained in a finite number of steps by application of the Armstrong rules of inference (see [reference 1 on page 257](#), [reference 2 on page 257](#), and [reference 11 on page 258](#) in the ["Bibliography" on page 257](#) and ["Rules of Inference for Deriving Additional FDs" on page 225](#)) to the given set.

Domain of a Data Element

The set of possible values that can be assigned to the data element. Different data elements may take on values from the same domain.

DSR-MVD

Abbreviation for DesignManager Multivalued Dependency. See "[Multivalued Dependency \(MVD\)](#)" on page 251.

Elemental FD

An elemental FD is one which has a single data element on its right-hand side. During the MERGE command, any FDs (from a USERVIEW being processed) with composite right-hand sides are decomposed into elemental FDs before being added to the workbench design area (see "[Merging Userviews into the Workbench Design Area](#)" on page 130).

Entity

Any object or concept about which data or information can be recorded. The data, when recorded, is at the level of properties or characteristics which describe the entity. These properties are called *attributes* and are represented in DesignManager by data elements. (Further details can be found in *ASG-DesignManager Enterprise Modeling*, available to users with the optional facility, Selectable Unit DSR-EM10, installed.)

Equivalence of Data Elements

If A and B are sets of (one or more) data elements and if, further, each of the FDs, $A \rightarrow B$ and $B \rightarrow A$, appears in or is derivable from the FDs of the composite view, then A and B are said to be equivalent. Such a pair of FDs constitutes a bijection.

Equivalence of FDs

A set F1 of (one or more) FDs is equivalent to a set F2 of FDs, where F1 and F2 need not be disjoint, if each FD of F1 is derivable from the FDs of F2 and each FD of F2 is derivable from the FDs of F1.

Extraneous Data Element

A data element A appearing in the (composite) left-hand side of a given FD of a composite view is extraneous if the FD obtained by eliminating A is derivable from all the FDs (including the given one) in the composite view. Such a data element is removed during processing of the DESIGN command (see "[The DesignManager Design Procedure](#)" on page 138). A data element appearing in both the left-hand and right-hand sides of a dependency (FD or MVD) is extraneous to the right-hand side. It is removed from the right-hand side during processing of the MERGE command (see "[Merging Userviews into the Workbench Design Area](#)" on page 130).

Fagin-MVD

See "[Multivalued Dependency as Defined by R. Fagin \(Fagin-MVD\)](#)" on page 251.

FD-Group

A group of dependencies formed in the workbench design area during the DesignManager design procedure (see "[The DesignManager Design Procedure](#)" on page 138, "[Step 3: Partitioning of the Composite View](#)" on page 141 and "[Step 4: Merging Groups with Identical Potential Keys](#)" on page 141) which contains one or more FDs all having the same left-hand-side and possibly one or more MVDs. A relation generated from an FD-group during the design procedure is termed an *FD-relation* and has as its key the left-hand side common to the FDs in the group.

FD-Relation

A relation generated by the DesignManager design procedure (see "[The DesignManager Design Procedure](#)" on page 138 and the DESIGN command specifications) from an FD-group.

First Normal Form

A relation is in first normal form if the domain of each of its data elements is restricted to simple values.

Foreign Key

A set of one or more data elements of a relation is a foreign key if it is the primary key of another relation, called the *target* relation.

Foreign Key Association

A unidirectional association between two relations in the workbench design area indicating the existence of a non-trivial FD from the key of one relation (the source relation) to the key of the other relation (the target relation). The key of the target relation appears as a foreign key (including at least one non-prime data element) in the source relation. A foreign key association is said to hold from the source relation to the target relation. A foreign key association existing from relation R1 to relation R2 is called *direct* and is written:

R1 -----> R2

if there is no third relation R3 such that both:

- A foreign key association exists from R1 to R3, and
- The key of R2 is contained in the key of R3 (see ["Hierarchical Association" on page 249](#)).

For example, a direct foreign key association, R1 -----> R2, holds from R1(A,B), with key A, to R2 (B,C), with key B; also from R1(A,D,B), with key A+D, to R2 (A,B,E), with key A+B. However, the foreign key association from R1 to R2 is not direct for R1(C,A,B) with key C, R3(A,B,D) with key A+B, and R2 (A,E) with key A. Instead, the direct foreign key association, R1 -----> R3, plus the direct hierarchical association, R3 <<----> R2 is presented.

Direct foreign key associations are depicted in the output of both the PLOT LOGICAL-SCHEMA command and the PLOT NETWORK command (see ["Logical Schema Cluster Plot" on page 210](#) and ["Network Cluster Plot" on page 214](#)).

Fourth Normal Form (4NF)

A relation R is in 4NF if it is in BCNF and no non-trivial Fagin-MVD (other than an FD) holds in R. A schema is in 4NF if each of its relations is in 4NF.

Full Functional Dependence

A data element B is fully functionally dependent on a set of data elements A (which does not include B) if B is functionally dependent on A but not on any subset of A.

Functional Dependency (FD)

A functional dependency A -----> B holds from A to B (where A and B are data elements or sets of data elements) if each value of A determines exactly one value of B. B is said to be functionally dependent on A.

Hierarchical Association

A bidirectional association existing between two relations in the workbench design area if the key of one relation is contained in the key of the other. No explicit dependency between the keys need exist in the workbench design area. (The trivial FD which holds from the containing key to the contained key is derivable automatically during the design procedure and need not be entered by the user. Indeed, even if entered in a userview, it would not be MERGED into the workbench design area; see ["Merging Userviews into the Workbench Design Area" on page 130](#).) A hierarchical association existing between relations R1 and R2 (for which, say, the key of R1 is contained in the key of R2) is *direct* and is written:

R1 <---> R2

if there is no third relation R3 such that both:

- The key of R1 is contained in the key of R3.
- The key of R3 is contained in the key of R2.

For example, a direct hierarchical association, R1 <---> R2, holds between R1(A,D), with key A, and R2 (A,B,E), with key A+B. On the other hand, the hierarchical association between R1 and R2 is not direct for R1 (A,D) with key A, R3(A,B,E) with key A+B, and R2 (A,B,C,E) with key A+B+C. Instead, the two direct hierarchical associations, R1 <---> R3 and R3 <---> R2 are presented.

Direct hierarchical associations are depicted in the output of both the PLOT LOGICAL-SCHEMA command and the PLOT NETWORK command (see ["Logical Schema Cluster Plot" on page 210](#) and ["Network Cluster Plot" on page 214](#)).

Implied Functional Dependency

If a data element, which appears in a dependency described in the workbench design area, is a GROUP member of the modeling dictionary and another data element in the workbench design area is one of the CONTAINED members, then DesignManager automatically generates an *implied* FD from the GROUP data element to the CONTAINED data element, and unless it already appears there, adds the implied FD to the workbench design area (see [remark 7 on page 81](#) of ["MERGE" on page 79](#)).

Inconsistent Userviews

If an FD described in a USERVIEW data definition has the same left-hand side and right-hand side as an MVD in the same or another USERVIEW, the USERVIEW or set of USERVIEWS is said to be inconsistent (see [remark 6 on page 81](#) of ["MERGE" on page 79](#)).

Independence of Data Elements

Two disjoint sets B and C of data elements of a relation R are independent (orthogonal) with respect to a third set of data elements A of R if, in every table of values constituting an instance of R, when to a given value of A there corresponds m values of B and n values of C, the table for R must have mn rows containing the given value of A, corresponding to every distinct combination of the associated values of B and C. This is a fundamental concept in the definition of a Fagin-MVD (which can be used as the basis for the definition of 4NF), applicable when all the data elements of R are in A, B, or C.

Instance of a Relation

See ["Relation" on page 253](#).

Intersecting Data Element

If a data element appears as the right-hand side of more than one FD described in the workbench design area and is not contained in the left-hand side of any, it is termed an *intersecting data element*. (See further in ["Merging Userviews into the Workbench Design Area" on page 130](#).)

Join

Given (instances of) relations $R(X,Y)$ and $S(Y,Z)$, where X , Y , and Z are disjoint sets of data elements, then the join of R and S (written $R*S$) is the relation obtained by forming all possible rows with values x , y , and z , where x and y form a row of R and where y and z form a row of S (for the same value y in each). This is often referred to as a *natural* join.

Key

A set A of one or more data elements in a relation R is a key of R if every data element of R is functionally dependent on A , and if no subset of A has the same property. Every relation has at least one key; for, if no other set qualifies, then the entire set of data elements must be the key and the relation is termed *all-key*. If R has more than one key, each is termed an alternative key of R .

Left-Hand Side (LHS)

If either an FD $X \text{ ----> } Y$ or an MVD $X \text{ ---->> } Y$ holds from X to Y (where X and Y are data elements or sets of data elements), then X is called the left-hand side of the dependency and Y the right-hand side.

Logical Database Design Model

A conceptual description of the structure of a database; in DesignManager, a 3NF relational schema.

Many Pointer

See ["Pointer" on page 252](#).

Multiplicity of a Dependency

The average number of values (or sets of values) determined for the right-hand side of a dependency by a given value of the left-hand side. So, the multiplicity of an FD is necessarily 1.

Multivalued Association

A unidirectional association between two FD-relations in the workbench design area indicating the existence of an MVD from the key of one relation (the source relation) to the key of the other relation (the target relation). A multivalued association is said to hold from the source relation to the target relation. If a multivalued association holds from relation $R1$ to relation $R2$, it is by construction *direct* and is written:

$R1 \text{ ---->> } R2$

Multivalued associations are depicted in the output of the PLOT NETWORK command (see ["Network Cluster Plot" on page 214](#)).

Multivalued Dependency (MVD)

A multivalued dependency $A \twoheadrightarrow B$ holds from A to B (where A and B are data elements or sets of data elements) if, for each value of A, a variable number of values of B may be determined (possibly none). B is said to be multiply dependent on A. This is a less restrictive definition of MVD than that given by Fagin (see [reference 7 on page 257](#) of the ["Bibliography" on page 257](#) and this Glossary), and it is considerably easier for the user in formulating an MVD. The more restrictive form of MVD is referred to here as a Fagin-MVD. The less restrictive form is called simply an MVD (or, where required to avoid confusion, a DSR-MVD).

Multivalued Dependency as Defined by R. Fagin (Fagin-MVD)

A multivalued dependency $A \twoheadrightarrow B$ in the sense of Fagin (see [reference 7 on page 257](#) of the ["Bibliography" on page 257](#)) is a DSR-MVD (see MVD definition) that satisfies these additional conditions:

- It must be formulated in the context of a relation $R(A,B,C)$ in which it holds (where A, B, and C are data elements or sets of data elements, and C may be the null set).
- If it holds non-trivially in R (C is not the null set), then (letting $R_1(A, B)$ be the projection of R on A and B, and $R_2(A, C)$ be the projection of R on A and C) it must be true that the join of R_1 and R_2 results again in R, that is, $R_1 \Join R_2 = R$ (otherwise, the join would result in a loss of information). Equivalently, it can be said that B and C are independent or orthogonal in R with respect to A. That is, in any table of values constituting an instance of R, if to a given value of A there corresponds m values of B and n values of C, then the table must contain mn rows for the given value of A, corresponding to every combination of values of B and C.

A Fagin-MVD provides a necessary and sufficient condition for a relation to be decomposable into two of its projections without loss of information (see [Appendix, "Technical Background," on page 221](#)). It is felt, however, that such a restrictive definition of MVD is unrealistic, preventing a user, for example, from specifying the MVD SUPPLIER- NO \twoheadrightarrow PART-NO if the user anticipates the relation $R(\text{SUPPLIER-NO}, \text{PART-NO}, \text{QUANTITY})$ (representing the FD $\text{SUPPLIER-NO} + \text{PART-NO} \twoheadrightarrow \text{QUANTITY}$). However, in a physical implementation, the user may require the MVD SUPPLIER- NO \twoheadrightarrow PART-NO, perhaps for purposes of secondary indexing. DesignManager will permit the user to do this. The only purpose of such a restriction is, in reaching 4NF (or a weak form of 4NF), to prevent the loss of information in decomposing a relation. However, the same result (valid decomposition of relations) can be obtained more readily via the DesignManager procedure if the qualifying condition (non-loss of information) is not tied to the definition of an MVD, but is applied instead after the generation of 3NF relations. (See [Appendix, "Technical Background," on page 221](#).) Elimination of orthogonality considerations from the formulation of a DSR-MVD is particularly helpful when specifying MVDs for a large data base prior to generating relations. In contrast, the accurate specification of Fagin-MVDs at this point could be prohibitively difficult. A more detailed discussion of the advantage gained by using the DesignManager definition of MVD is found in the [Appendix, "Technical Background," on page 221](#).

MVD-Group

A group of dependencies formed in the workbench design area during the DesignManager design procedure (see ["The DesignManager Design Procedure" on page 138](#), Steps 3 and 4) which contains one or more MVDs but no FDs. Each of the MVDs of an MVD-group is composed of exactly the same data elements. A relation generated from an MVD-group is termed an *MVD-relation* and has as its key all the data elements appearing in the group.

MVD-Relation

A relation generated during the DesignManager design procedure (see ["The DesignManager Design Procedure" on page 138](#) and the DESIGN command specifications) from an MVD-group. An MVD relation is also termed an *all-key* relation (see ["Key" on page 250](#)).

Non-prime Data Element

See ["Prime Data Element" on page 253](#).

Nontrivial MVD

See ["Trivial MVD \(in a Relation\)" on page 256](#).

Normalization

Applying those steps of the DesignManager design procedure to a composite view which ensure generation of relations in 3NF (see ["The DesignManager Design Procedure" on page 138](#)).

Normalized Schema

In DesignManager, a schema which is in 3NF.

One Pointer

See ["Pointer" on page 252](#).

Orthogonality of Data Elements

See ["Independence of Data Elements" on page 249](#).

Pointer

A pointer from one normalized relation in the workbench design area to another indicates the type of association holding between the two relations. Two types of pointers are defined, *one* pointers and *many* pointers. A *one* pointer is defined from relation R1 to relation R2 if the key of R2 is a foreign key in R1 (possibly as a subset of R1's key) and there is no other relation R3 such that both:

- The key of R3 is a foreign key in R1.
- The key of R2 is contained in the key of R3.

A *many* pointer is defined from relation R1 to relation R2 if the key of R1 is contained in the key of R2 and there is no other relation R3 such that both:

- The key of R1 is contained in the key of R3.
- The key of R3 is contained in the key of R2.

Pointers between relations are indicated in the output of the REPORT LOGICAL-SCHEMA command. See ["The Logical Schema Report" on page 116](#) for further discussion and examples.

Potential Key

A set of data elements appearing in the dependencies of an FD-group or an MVD-group which would be the key of a relation generated from the group by the DesignManager design procedure (see ["The DesignManager Design Procedure" on page 138](#), ["Step 3: Partitioning of the Composite View" on page 141](#) through ["Step 5: Generation of 3NF Relations" on page 143](#)). The potential key of an FD-group is the left-hand side common to all the FDs in the group. The potential key of an MVD-group is the set of all the data elements appearing in any of the dependencies of the group (each MVD of an MVD-group is composed of exactly the same data elements).

Primary Key

Each relation of a schema has at least one key. To satisfy users' primary access needs, one of the keys is designated as the primary key, although in practice the relation can be sorted on any other set of data elements for alternative access requirements.

Prime Data Element

A data element is prime in a relation R if it is a key of R or is contained in any key of R. Otherwise, it is non-prime.

Projection

The projection of a relation R on a subset X of its data elements is the relation obtained by removing all columns of R labeled with data elements not in X and then removing any duplicated rows in the resulting table.

Pseudo-FD

A pseudo-FD $A \multimap B$ from data element A to data element B states that A is a subcategory of B and thus takes on values from a subset of the domain of B. It can be read *is a B*; that is, *each value of A is also a value of B*. See ["Use of Pseudo-FDs to Handle Subcategories" on page 151](#).

Redundant FD

An FD of a composite view is redundant if it is derivable from the set of FDs in the composite view.

Relation

A logical description of a set of data elements. The structure of a relation R is a list of data elements A_1, A_2, \dots, A_n (written $R(A_1, A_2, \dots, A_n)$) and a set of dependencies (FDs and/or MVDs) holding among them. In a physical implementation, an instance of a relation is a two dimensional table of values defined over a set of data elements, such that:

- Each column of the table is labeled with the name of a distinct data element and contains values from the domain of the data element.
- No two rows are identical.

Relational Database Model

A set of relations defined over the data base data elements. In DesignManager, the relations are generated from input userviews of the data elements. Each userview is expressed as a set of dependencies required by a user for a particular application.

Relational Schema

A relational database model satisfying these conditions:

- Each relation in the model must have a key.
- All access paths indicated by the dependencies of the composite view (from which the schema was generated) must be present in the relations; that is, each input dependency must be represented either directly in a single relation or indirectly via several relations.

Relation Number

A number is automatically assigned by DesignManager during the processing of a DESIGN command (see the ["DESIGN" on page 74](#) command specifications) to identify a relation which has been generated in the workbench design area. This number may be specified by a user when naming the relations in a subsequent NAME command (see the ["NAME" on page 82](#) command specifications). It is also used in the output produced by various design mode reporting commands.

Relative Dependency Number

See ["Absolute Dependency Number" on page 245](#).

Relative Frequency

The number of times the process represented by a userview is invoked per unit time in the database.

Response Time

The acceptable time interval from the moment data is requested from the data base by the process represented by a userview to the moment the data is received by the process.

Right-Hand Side (RHS)

See ["Left-Hand Side \(LHS\)" on page 250](#).

Role

Given the pseudo-FD $A \text{ ----} \rightarrow B$ (that is, the data element A is a subcategory of the data element B), then A is a role played by B, if those values of B which are values of A at one instant of time (or in one context) need not be values of A at another instant of time (or in another context). For example, a FIRM may be a BUYER-FIRM in one transaction and a SELLER-FIRM in another; at other times, it may be neither.

Schema

See ["Relational Schema" on page 253](#).

Secondary Key

In a physical implementation of a schema, a data element (or set of data elements) may be designated as a secondary key (to satisfy a specific access requirement) if it multiply determines the data elements of the same or another relation, ordinarily through the primary key.

Secondary Key Association

A unidirectional association between two FD-relations in the workbench design area indicating the existence of an MVD from a non-key set of data elements in one relation (the source relation) to the key of the other (the target relation), except that the non-key set of data elements in the source relation cannot contain the key of the source relation nor can it be a foreign key to any other (third) relation. A secondary key association is said to hold from the source relation to the target relation. If a secondary key association holds from relation R1 to relation R2, it is by construction *direct* and is written:

R1 ---->> R2

For example, a secondary key association, $R1 \dashrightarrow R2$, holds from $R1(A,C)$, with key A , to $R2(B,D)$, with key B , if an MVD from C to B is present on the workbench (unless there is a third relation $R3$, with key C , on the workbench, in which case these associations would hold instead:

- A direct foreign key association from $R1$ to $R3$.
- A multivalued association from $R3$ to $R2$, based on the MVD from C to B).

As a second example, a secondary key association would hold from $R3(A,E,C)$, with key $A+E$, to $R4(F,D)$, with key F , given an MVD from E to F .

More than one secondary key association can hold from a source relation to target relation. In the second example above, any of the following MVDs would lead to an additional secondary key association from $R3$ to $R4$:

$A+C \dashrightarrow F$, $E+C \dashrightarrow F$, or $A \dashrightarrow F$

One or more secondary key associations can hold from a relation to itself, for example, $R(A,B,C)$ with key A , where:

$C \dashrightarrow A$ and/or $B \dashrightarrow A$

Secondary key associations are depicted in the output of the PLOT NETWORK command (see ["Network Cluster Plot" on page 214](#)).

Second Normal Form (2NF)

A relation is in 2NF if each of its non-prime data elements is fully functionally dependent on each of its keys. A schema is in 2NF if each of its relations is in 2NF.

Structure of a Relation

The set of data elements over which the relation is defined plus the functional and multivalued dependencies which hold among them.

Subcategory of a Data Element

A data element A is a subcategory of another data element B if the domain of A is a subset of the domain of B and if, further, any data element which is functionally dependent on B is also functionally dependent on A .

Target Relation

See ["Foreign Key" on page 248](#).

Third Normal Form (3NF)

A relation is in 3NF if none of its non-prime data elements is transitively dependent on any of its keys. If a relation is in 3NF, then it must be in 2NF. An all-key relation is automatically in 3NF. A schema is in 3NF if each of its relations is in 3NF.

Total Access Frequency (TAF)

Total access frequency is accumulated, for a data element represented in the workbench design area, over all userviews described in the workbench design area as follows: TAF is initially set to zero; then, in each userview, it is incremented over every path of dependencies leading to the data element, by the appropriate USERVIEW RELATIVE-FREQUENCY specification multiplied by the product of the multiplicities of the dependencies in the path. If, in any userview, the data element has no path leading to it but has at least one path emanating from it, then the contribution to TAF in that userview is just the value of RELATIVE-FREQUENCY specified for the userview.

Transitive Dependence

If A and C are data elements or sets of data elements of a relation R, then C is transitively dependent on A (in R) if there exists another data element (or set of data elements) B in R, disjoint from C, such that both of the FDs, $A \rightarrow B$ and $B \rightarrow C$, hold in R (but not $B \rightarrow A$).

Trivial FD

An FD is trivial (it must always hold) if all the data elements of the right-hand side of the FD are also contained in the left-hand side. Otherwise it is nontrivial. For example, $A \rightarrow A$ and $B + C \rightarrow C$ are trivial; that is they hold for any A, B, and C.

Trivial MVD (in a Relation)

An MVD is trivial in a relation R (it must hold in R) if every data element appearing in the MVD is a data element of R and every data element of R appears in the MVD. An MVD is non-trivial in R if every data element appearing in the MVD is a data element of R, but at least one of the data elements of R does not appear in the MVD. For example, $A \twoheadrightarrow B$ is nontrivial in $R(A,B,C)$, whereas $A + B \twoheadrightarrow C$ is trivial in R (it holds in R for any A, B, and C).

Userview

A set of dependencies (functional and/or multivalued) which express a user's requirements for accessing the values of the database data elements in a particular application.

Userview Number

See ["Data-view Number" on page 246](#).

Viewset

A collection of userviews, normally identifying input for generation of a schema.

Weighted Average Response Time

Weighted average response time is calculated, for a data element represented in the workbench design area, over all the occurrences of the data element on any path leading to it in any userview described in the workbench design area (see ["Total Access Frequency \(TAF\)" on page 256](#)). The weighting factor applied in each userview (to the appropriate RESPONSE-TIME specified for the userview) is the component of the total access frequency calculated for the data element in that view.

Bibliography

1. Beeri, C., and Bernstein, P. A. *Computational Problems Related to the Design of Normal Form Relational Schemas* ACM Transactions on Database Systems, Vol.4, No.1. March 1979, pages 30-59.
2. Beeri, C., Bernstein, P. A., and Goodman, N. *A Sophisticate's Introduction to Database Normalization Theory* Proceedings of the Fourth International Conference on Very Large Data Bases. Berlin, September 1978, pages 113-124.
3. Bernstein, P. A. *Synthesizing Third Normal Form Relations from Functional Dependencies* ACM Transactions on Database Systems, Vol.1, No.4, December 1976, pages 277-298.
4. Biskup, J., Dayal, U., and Bernstein, P. A. *Synthesizing Independent Database Schemata* Proceedings of the ACM SIGMOD Conference, Boston Massachusetts, May 1979, pages 143-151.
5. Chen, P. P. *The Entity-Relationship Model-Toward a Unified View of Data* ACM Transactions on Database Systems, Vol.1, No.1, March 1976, pages 9-36.
6. Codd, E. F. *Further Normalization of the Data Base Relational Model* In Data Base Systems, Courant Computer Science Symposia Series, Vol.6, Prentice-Hall, Englewood Cliffs, New Jersey, 1972, Pages 33-64.
7. Fagin, R. *Multivalued Dependencies and a New Normal Form for Relational Databases* ACM Transactions on Database Systems, Vol.2, No.3, September 1977, pages 262-278.

8. Zaniolo, C. *A New Normal Form for the Design of Relational Database Schemata* ACM Transactions on Database Systems, Vol.7, No.3, September 1982, pages 489-499.

In reference 8, a new normal form is presented, Elementary Key Normal Form (EKNF), which lies between Third Normal Form (3NF) and BoyceCodd Normal Form (3CNF); that is, every 3CNF relation is in EKNF and every EKNF relation is in 3NF. The new normal form provides all the advantages of both 3NF and 3CNF, while avoiding the shortcomings of the latter. It results in a more favourable separation of data values than 3NF in a physical implementation, and thus a further reduction of updating problems. Furthermore, EKNF is compatible with the principle of *faithful representation* (see ["Keys and Representation of FDs in a Schema" on page 227](#)) of input dependencies (when relations are *synthesized* from a set of specified dependencies, as in DesignManager) and, in a synthesis procedure, it requires no more computation time than does 3NF. A 3CNF schema, on the other hand, does not necessarily represent all specified dependencies and computationally can be very difficult and time-consuming to generate. It is of particular interest to users of DesignManager that, in reference 8, Zaniolo proves that the Bernstein design algorithm, which is the basis for the DesignManager design procedure, produces relations which are automatically in EKNF as well as in 3NF.

9. Fagin, R. *The Decomposition Versus the Synthetic Approach to Relational Database Design* Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, October 1977, pages 441-446.
10. Codd, E. F. *Recent Investigations in Relational Database Systems* Information Processing 74, North-Holland Publishing Company, Amsterdam, 1974, pages 33-36.
11. Beeri, C., Fagin, R., and Howard J. H. *A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations* Proceedings of the ACM SIGMOD Conference, Toronto, Canada, 1977, pages 47-61.
12. Date, C. J. *An Introduction to Database Systems* Third Edition, Addison- Wesley Publishing Company, Reading, Massachusetts, 1981.
13. Fagin R. *Normal Forms and Relational Database Operators* Proceedings of the ACM SIGMOD Conference, Boston, Massachusetts, May 1979, pages 153-160.
14. Aho, A. V., Beeri, C., and Ullmann, J. D. *The Theory of Joins in Relational Databases* Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, October 1977, pages 107-113.

Symbols

`:-` 183
`:+` 183
`:ADD` 184–185
`:BOTTOM` 183
`:CHANGE` 184–185
`:COPY` 184–185
`:DELETE` 184–185
`:DOWN` 183
`:DUPLICATE` 184–185
`:FAIL` 186
`:INSERT` 184
`:LOCATE` 183–184
`:MACRO` 186
`:MEND` 186
`:-n` 184
`:NUMBERS` 183–184
`:NUMBERS OFF` 184
`:POSITION` 183–184
`:PROMPT` 186
`:READ` 186
`:SCHANG` 184
`:SET` 183
`:SHIFT` 184–185
`:SKIP` 186
`:SWITCH` 183–184
`:TOP` 183–184
`:UP` 183–184
`:VERIFY` 184–185
`:WIPE` 184–185
`:WRITE` 186

A

abbreviation of keywords
 see truncation of keywords
access frequency
 see relative frequency
access requirements, of database
 see Dependencies, Userviews
access security, of modeling dictionary 26
ACCESS-AUTHORITY clause 99
ADD command 8, 27, 98, 146–147

 specification 28
ADMINISTRATIVE-DATA clause 100
ALIAS clause 99, 126, 175
 specification 101
aliases 7, 27, 175
 name length 20
 see also LIST command specification
 types 175–176
all-key relations 129, 136, 141, 228
 see also MVD-relations
alternative key 228
amendment lines 17, 27
 see also MODIFY command
ampersand (&) command
 specifications 180, 187
associated userviews 115
associations between relations 210
audit trail of design procedure 75
augmentation rule 226
AUTHORITY command 11, 25–26
 specification 31
automatic error recovery 30

B

batch/interactive, input differences
 see interactive batch input differences
BOMP 157, 161–162
 see parts explosion problem
Boyce-Codd normal form 221, 232
BULK command 173

C

candidate key
 see alternative key
catalogue classifications 7, 27, 127
 see also LIST command specification
CATALOGUE clause 126
 specification 101
character strings 21
clauses, in statements 16
CLEAR command 71, 74
 specification 73

- closure 229
- cluster of relations 210
- CMS environments 34
 - interface facility 165
- command identifier 16, 23
- command statements 15
 - see also* dictionary mode, design mode, and mode command
- COMMAND-STREAM member type 145, 172
- COMMENT clause specification 102
- common clauses 90–92, 99
- completeness 222
- composite 130
- composite key
 - see* concatenated key
- composite view 123, 130, 132–133, 138–139
- concatenated key 154, 228
- CONTAINS clause 29, 93, 97–98
 - use in top-down methodology 130
- Controller 11, 23–24, 26
- conventions page viii
- COPY command 10, 26–27
 - specification 32
- covering 227
- D**
- data definition statements
 - see* Member definition statements
- data element analysis report 113, 115
- data elements 2–4, 7, 87, 124
 - and synonyms 12
 - identification of 3
 - in dependencies 6, 130
 - naming of 124–125
- data entries dataset 6–7, 27–29, 92–93, 96, 98
- data entries datasets 35
- databases, physical 2, 12
 - problems with 1, 134
- DataManager and integrated environments 169
- data-view report 113
- data-views 71, 73
- DCUST macro 34, 103–105
- decimal numbers 19
- decomposition (of relations) 221
- decomposition rule 226
- dependencies 5, 72–73, 123
 - and normalization 132
 - determination of 128
 - see also* functional dependency, multivalued dependency
- dependency number
 - absolute 78, 80
 - relative 80
- DESCRIPTION clause 126
 - specification 103
- design audit report 120
- DESIGN command 113–114, 129–130, 132–133, 138, 144, 167
 - statement specification 74
- design mode commands 13, 73
 - effects of integration with DataManager 169
 - output 7
- design procedure 72, 133
 - see also* DESIGN command specifications, Normalization
- design reports 113
- DesignManager 1
 - and logical database design 1
 - safety and control features 12
- detail report 118, 120
- DICTIONARY command 11, 25–26
 - specifications 33
- Dictionary mode 13
 - commands 15, 23, 28
 - effects of integration with DataManager 170
 - function 23
 - output 112
- Dictionary *see* Modeling Dictionary
- domain 134, 151, 157, 160
- DSR-MVDs 222
- dummy data entries records (dummy members) 62, 69, 93, 96, 98, 108, 129
 - creation as a warning 12
- E**
- ECHO command 73
 - specification 75
- echoed input lines 111
- EFFECTIVE-DATE clause
 - specification 103
- elemental functional dependencies 78, 81, 131
 - see also* functional dependencies
- elements (in statements) 17
- embodiment (of FDs) 229
- encoded records
 - see* data entries dataset
- ENDDSR command 26, 31, 33
 - specification 37, 76
- enterprise view 133
- entities 133
- entity report 167
- entity-relationship model 133

- ENVIRONMENT command 25
 specification 38
 error recovery dataset 10, 30
 errors
 see messages
 ESCAPE character 180–181
 EXECUTE command 182
 extraneous data elements 81, 131, 139, 226,
 229, 238–239
- F**
 Fagin-MVDs 221, 234–237, 243
 faithful closure 222, 229
 faithful representation 229, 232, 234–235
 FD-groups 81, 141–142, 240
 FD-relations 75, 141, 143
 FETCH command 71–72, 145
 specification 76
 first normal form 230
 floating point numbers 16, 19–20
 foreign keys 154, 228
 FORMAT members 166, 176
 format-selection clause 75, 166, 176
 see also REPORT command
 specification, User Formatted
 Output
 fourth normal form 221, 230, 234, 237
 FREQUENCY clause specification 104
 functional dependencies 128
 functional dependency 4, 78, 80
 determination of 124, 129
 full 230
 see also MERGE, DESIGN command
 specifications
- G**
 GLOSSARY command 173
 GROUP member type 24, 81–82, 126, 175
 data definition 92
 use in top-down methodology 130
- H**
 history of members' records
 see LIST command specification
 homonyms 126, 132, 155
- I**
 ICCF Interface facility 165
 implied functional dependencies 81, 131,
 225
 inconsistent data-views 71, 73, 87
 inconsistent multivalued dependencies 13,
 87
 inconsistent userviews 75
 incremental design procedures 130, 133
 independent data elements 234
 index dataset 6–7, 27, 47, 62–63, 90
 inference rules 225, 238
 input lines 16, 109
 numbers 111
 instance (of a relation) 229
 integers 19
 integrated environments 24, 112, 128, 145
 integration facility 165, 169
 specification of 170, 175
 interactive Front-end facility 179, 182
 interactive/batch, input differences 16, 18,
 72
 see also interactive front-end facility
 intersecting data elements 132
 see also REPORT command
 specification
 intersecting data elements report 118–119
 ITEM member type 89, 91, 126, 175
 data definition 90
 use in top-down methodology 130
- K**
 KEY clause *see* NAME command
 specification
 keys of relations 72, 133, 135, 231, 240
 see also KEY clause, primary keys,
 potential keys
 keywords 16
- L**
 LBUF1 macro 180–181
 line numbering, in source records 28
 see also MODIFY, ADD, PRINT
 command specifications
 line printer output 109–111
 LIST command 77, 127, 130, 132
 effect of integration with
 DataManager 112, 174
 specification 38, 77
 load factor calculation facility 95, 167
 load factor calculation report 113, 167
 logical database design models
 see relational schema
 logical schema report 116, 118
 logical schema *see* relational schema
 LOPT1 macro 19, 110, 180, 183, 204
 lossless join property 221, 234
- M**
 macro usage commands 186

magnetic tape/disk output 109
Manager Products Controller's Manual 24
Manager Products Message Guide 110
master operator 31
member definition statements 15, 28, 89–90
member names 90–91
member type identifiers 89
MERGE command 130, 132–133, 144, 171
 effects of integration with
 DataManager 170
 specification 79
 use and action 130, 144
messages 109–110
MODE command specification 25
mode commands 23
modeling dictionary 6
 access security commands 26
 automatic error recovery of 10
 controller actions 10–11
 documentation commands 27
 manipulation commands 26
 member types 7
 name length 19
 records 8
 see also DICTIONARY command
 statement specification
 structure 6
modes 13, 23
 see also Dictionary mode, Design
 mode
MODIFY command 10, 26–27, 89, 147
 specification 44
multiplicity 5, 94, 129
multivalued dependency 4, 223
 determination of 124
 processing by MERGE command 131
MVD-groups 75, 80, 141
MVD-relations 128, 133, 136, 144
 merging with FD-relations 130

N

NAME command 71–72, 141, 144, 151
 specification 82
 use of 145
name-related-selection clause 63
 see also LIST command specification
 specification 63
naming standards 126
network cluster plot 214
NO-ECHO command 73
 specification 84
non-key data elements 228
 see also keys of relations
non-prime data elements 228

see also prime data elements
normal forms 230
 see also first normal form, second
 normal form, third normal form,
 Boyce-Codd normal form,
 fourth normal form
normalization 123
 benefits of 133
 see also design procedure
normalized/unnormalized data 72–73
 see also DESIGN, MERGE command
 specifications
notation for statement formats ix
NOTE clause 126
 specification 105
nucleus 165

O

OBSOLETE-DATE clause
 specification 105
OBSOLETE-DATE keyword 91–92, 95, 97
optional additional facilities 23–24, 165
 effects of integration with
 DataManager 175
optional-additional-member-type 175
orthogonal data elements
 see independent data elements
OS environment 34
output formats 109
 see also User Formatted Output
 facility

P

page throws in output 110
partitioning 240
parts explosion problem 151, 157
 see also pseudo-FDs
passwords 11, 26, 31–32
PERFORM command 145, 171–172, 174
PLOT command 133
 specification 209
pointers 30
pointers between modeling dictionary
 members 9, 30, 46–47, 96, 98
pointers between relations 117
potential keys 141, 240
PREFIX character 183, 187, 204
primary keys 79
prime data elements 228
PRINT command 126
 effect of integration with
 DataManager 112
 specification 60

- use in naming data elements 132
- process DataManager member types 175
- pseudo-FDs 151
- pseudotransitivity rule 226
- punctuation (in statements) 17
- Q**
- QUERY clause specification 106
- QUERY keyword 91–92
- R**
- records, of modeling dictionary members 9
- redundant functional dependencies 87, 136, 140, 225, 227
- reflexivity rule 226
- relation scheme 223
- relational schema 2, 71, 123, 133, 222
 - advantages of 133
 - see also* DESIGN, REPORT command specification
- relations 2, 86, 138, 222
 - all-key 144
 - generation of 138
 - keys 135
 - naming of 141
 - number 85, 87
 - storing as userview members 144
- relative frequency 5, 95, 129
- RELATIVE-FREQUENCY clause 95
- REMOVE command 10, 26–27
 - specification 61
- report category 113
- REPORT command 112–113, 130, 132
 - see also* user formatted output facility specification 84
- REPORT command (DataManager) 176
- representation (of FDs) 227
- response time 5, 94, 124
- restricted commands 23
- role relations 152
- roles 152–153
- rules governing variables 19
- S**
- safety and control features 12
- second normal form 136, 221, 231
- security system 11
 - see also* access security commands
- SECURITY-CLASSIFICATION clause
 - specification 107
- SEE clause specification 108
- selectable unit
 - see* optional additional facility
- SKIP command 171
- SNAPSHOT command 72–73, 130
 - specification 86
- source dataset 6–8, 27, 30, 32, 46, 48–49, 58–59, 92, 96, 98
- source record
 - see* source dataset
- space characters, in statements 20
- SPACE command 171
- statement body 15
- statement identifiers 15–16
- statements
 - notation for
 - see* notation for statements
 - used in batch 16
 - used interactively 16
- STORE command 72, 145–146
 - specification 87
- summary report 120
- superkeys 229
- SWITCH command 145, 172–173
- synonyms 12, 125–127, 132
- synthesis procedure 231
- T**
- target relations
 - see* foreign keys
- terminators 15, 18, 90
- third normal form 2, 132, 221, 225, 230
 - see also* relational schema
- time-and-user-related-selection
 - clause 63
 - see also* LIST command specification
- time-and-user-related-selection clause
 - specification 67
- top-down methodology 130
- transitive dependence 231–232
- transitivity
 - see* redundant functional dependencies
- trivial functional dependency 131, 151, 226
- truncation of keywords 17
- TSO 37, 76
 - Interface facility 165
- U**
- union rule 226
- unverified/verified modeling dictionary
 - member records 72, 79–80
- update lock 10
- updating commands of modeling dictionary 7, 10
- user defined syntax facility 177

User Formatted Output facility 166, 209–210
 effects of integration with
 DataManager 176, 178
user names, length of 20
 see also member names, passwords
User Printer Graphics facility 209
 effects of integration with
 DataManager 176
userview report 113
userviews 2–3, 5, 13, 85, 87, 98, 123
 data definition 96
 development 125, 128
 for storing relations 144
 formulation by top-down
 methodology 130

V

validity 223
variables 17, 19
verified/unverified modeling dictionary
 member records 72, 79–80
VIEWSET 89, 125, 130
 data definition 97
visual display unit, output 109
 see also Interactive Front-end facility

W

WHAT command 174
WHICH command 174
workbench design area 85, 87, 160, 163, 209
 documentation commands 71–72
 manipulation commands 71
 percentage usage 87
workbench file 72, 87

ASG Worldwide Headquarters Naples Florida USA | asg.com