Part I: INTRODUCTION TO RDMS

Section 1: RDMS Concepts and the RDMS Environment

Section 2: The Modular Interface to the Data Base

Part II: RDMS Commands

The Relational Data Mangement System (RDMS) provides a generalized data-base management and reporting system.*. The data-base management system consists of a uniform method for cataloging files (called relations) and character string data, and a set of commands and subroutines (including a generalized computational facility) which operate on the relations of a data-base. In addition, RDMS provides an editor for entering and modifying data, commands for printing selected contents of a data-base and generating reports, and a language for report generation and data-base query.

A relation may be thought of as a computer-stored table consisting of a variable number of rows (called tuples) and and a fixed (for the relation) number of columns (called data-types). For example, a telephone book may be considered a relation of three columns (name, address, and telephone number) and as many rows as there are entries in the book. The table is called a relation because the information (datums) in each column (data-type) of a row (tuple) is associated with (related to) the information in the other columns. For example, the address and telephone number in each row of the telephone book are related to (associated with) the person whose name is listed in that row.

Each data-type in the relation has a meaning and method of displaying the information specific to that column. That is, an entry (datum) in the "name" data-type is of the form "Doe, John C.", a datum in the "address" data-type is of the form "26 Farm Road Lex.", and datum in the "telephone number" data-type is of the form "253-4107." The column is labeled with a data-type name, such as "name," "address," "telephone number," "title," etc. The relation/row/column concept is a refinement of the traditional file/record/field concept in which each column (field) may contain information of variable length.

Thus, an RDMS data-base is a collection of relations, each of which represents a set of relationships between data elements (datums) from different data-types. Two separate relations of the data-base may have a common column name, and although these relations are distinct, they are implicitly linked because they involve the same objects (datums of the same data-type or column name).

For this reason, if two (or more) relations have the same

---

* RDMS is currently implemented in PL/1 on the Multics operating system.

column names (data-type names), they can be combined (or operated on) to produce a third relation. Three standard RDMS operations are union, intersection, and difference. The union of two relations results in a relation consisting of all the rows in either of the original relations (without duplicate rows). The intersection of two relations results in a relation consisting of all the rows which are in both of the original relations. The difference of two relations results in a relation consisting of all the rows which are in the first original relation and not in the second.

The "compose" operation can be used to combine two relations with some column names (data-types) in common into a new relation. The "cartesian product" operation can be used to combine two relations which have no column names in common into a new relation. The "sort" and "project" operations act on a single relation to sort rows by column name or to eliminate columns, respectively.

In addition to these relation manipulation operations, RDMS also includes computation packages which greatly enhance the utility of the relational concept of data. The relation_operators package (rop) permits the user to perform a computation which reduces a number of groups of tuples into a set of tuples, one new tuple for each group of tuples. The groups of tuples are identified by dividing (partitioning) the tuples of a relation into subsets (blocks) based on the data in each of the relation's columns, starting from the leftmost. (For example, this permits the user to compute nested sums such as subtotals, totals and grand totals.) The user of the rop need specify only the data relation, a partitioning of it, and the algorithm to be performed to reduce each block of the partitioned relation into a single output tuple.

A command language exists which performs all the above operations on relations. A relation editor "eds" (edit set) is used to create relations and input, update, and maintain data in relational form.

There are a number of programs providing varying control over the number of relations which can be reported on and the format of the output. "quick_report" prints a single relation in tabular form. It allows the use of headers, footers, permutation of columns, user-supplied formats and the ability to ignore certain columns in printing, all with little effort by the user. "Book" and "display_relation" provide the capabilities of quick_report, and in addition permit the printing of relations while making formating and print decisions based on the semantics

of stepping through relations (which columns change, the leftmost column which changed, etc.). Thus, the format of the output can reflect the grouping of the data in the relation.

"Report" enables the user to access relations in terms of keys (datums) from other relations, thus providing the facility whereby a report may be generated based on data retrieved from several relations. Report is useful in keyed-query type reports and large reports which format their output based on the semantics of keyed retrieval from several relations simultaneously.

RDMS thus provides complete facilities for persons who must create, manipulate, and use data-bases. For further information on the implementation of data-types, the data strategy and format, and RDMS' dependence on features of Multics, see Section 1.6, RDMS Design Principles.

(END)

THE RDMS COMMAND REPERTOIRE

Most of the commands and topics mentioned below are documented elsewhere in this manual. This "road map" is intended to be useful in helping the new RDMS user to find the RDMS facility (s)he requires. Thus the list is not all inclusive, but should instead be regarded as a good starting point. Those topics documented outside of the RDMS REFERENCE GUIDE are followed by a note indicating where the documentation can be obtained.

In addition, the Table of Contents and the Index may be useful tools for locating information on a particular subject.

1) Creating and Initializing a Data Base

createdb.ec	Creates a multics directory and initializes it as a data base.
set_data_base	"Sets" an already existing data base (i. e., prepares it for subsequent use.)

2) Creating Data Types

new_data_type	Creates a new data type managed by a specified (user or RDMS supplied) data-strategy-module.
eds	When eds ("edit set") creates a new relation, it may also create new data types.
tree_stuff	Adds new data elements to an empty "astring" data type in the order which results in an optimal binary tree. (For data types managed by dsm_v2_astring or a similar dsm.)

3) Creating Relations

create_relation
: Creates an empty relation with specifed data types in specified sort order.

eds
: Creates a new relation or operates on a old relation. Eds accepts input which requests it to insert new data or modify existing data in a relation.

sms_interface
: A command level interface to the RDMS primitive operations. The operations use existing relations to produce a new relation. The primities include union, intersection, difference, composition, cartesian product, and adding tuples (rows) to a relation.

make_relation, make_quart
: Make an relation or quart with specified sort order.

4) Modifying the Contents of an Already Existing Relation

eds
: The relational editor can be used to add, delete, or update the data in a relation.

sms_interface
: This command level interface to the RDMS primitive operations can be used to make changes over an entire relation in one operation, in the sense that a new relation can be created and substituted for the old in one command.

restructure_data_type
: When garbage collection is performed to remove unused data from a data type, or a data type is converted from one strategy module to another, relations using that data type are automatically updated.

5) Listing/Printing the Objects in a Data Base

print_data_base
: This command prints the pathname of the currently set data base, or prints a message if no data base is set.

[dbd]
: This active function returns the path of the currently set data base, or if no data base has been set a null string is returned.

sms_star
: This command/active function prints or returns a list of objects in the data base which meet certain constraints as to their name or type.

set_name
: This command/active function prints/returns the list of names on a given object.

set_user
: This command/active function prints/returns the list of objects which use a specified object. Thus the list of relations in which a data-type occurs or the list of data-types managed by a certain data-strategy-module may be obtained.

list_sets
: Lists the names and reference numbers of objects in the data base, including all relations, data types, and strategy modules.

status_sets
: Gives all availible information about a relation, data type or quart. For relations, it can be used to obtain the length, order and data types used, while for data types it gives the number of data elements.

print_set
: Prints the rows of a relation or the data elements in a data type.

display_relation
: Prints the contents of a relation which control over the format of the output.

list_data_type
: Lists the data elements of a data type, along with their reference numbers.

eds	The relational editor can be used to print some or all of the tuples (rows) in a relation with some format control.

6) Data Base Maintainence

delete_set	Deletes a set. (i. e., a data-type or relation.)
copy_set	Copies a data type or relation from another data base.
insert_set	Links to a data type or relation in another data base so one copy can be used from either data base.
add_name_set	Adds a name to a data type or relation.
delete_name_set	Removes a name from a set.
rename_set	Changes the name on a set.
insert_module	Makes a new strategy module usable from a data base.
delete_module	Removes an strategy module from the data base.
status_sets	Gives all information availible about a set.
list_sets	Lists objects in data base, along with reference number and strategy module.

7) Report Generation from Relations

print_set	Simplest method of printing the contents of a relation. No formating.
eds	The relational editor can print the contents of a relation with some format control.
quick_report	Outputs each row of a relation with data types lined up in columns. Data format modules may be specified for the data types,

as well as the sort order in which the data types appear.

display_relation,
book — Allows more format control. If a data type (column) in a relation does not change in passing from one tuple (row) to the next, printing of that data type may be supressed. Additionally, vertical formating can be specified: "when the first data type changes, start a new page and print a header".

report — Procedural language which allows output from more than one relation at once, with the format of output depending on tests made during report production.

data format modules — Can be used to abbreviate, underline or otherwise format the appearance of data elements of a specified data type.

8) Running or Requesting Reports

exec_com — This Multics command is documented in the MULTICS PROGRAMMERS' MANUAL, Command Section. It is useful for placing a sequence of command lines in a file so that invoking exec_com with the name of the file has the same effect as typing the command lines. Since report production normally requires execution of a sequence of command lines, the process is can be expedited by the use of a ".ec" file containing the commands.

runoff — This Multics Command for formating text is documented in the Command Section of the MULTICS PROGRAMMERS' MANUAL.

dprint — Multics Command for printing files on offline high speed line printer. Documented in the MULTICS PROGRAMMERS' MANUAL, Command Section.

enter_absentee_request — This Multics command is useful for requesting a report production which will then be run at

the lowest availible rates. See the description of this command in the MULTICS PROGRAMMERS' MANUAL, Command Section.

cancel_absentee_request
: Multics command to cancel an absentee process request. Can be used to cancel a report request submitted using enter_absentee_request. See the MULTICS PROGRAMMERS' MANUAL, Command Section.

prologue
: A program for setting up standard headers and footers in reports. The relation "sms_prologue" must be in the data base; the header produced for a report depends on the information associated with the report name in sms_prologue.

9) Communication Between Users

mail
: Multics command for sending mail. (MULTICS PROGRAMMERS' MANUAL - Command Section)

send_message
: Multics Command which sends a message immediately printed on recipient's console. (MULTICS PROGRAMMERS' MANUAL - Command Section)

whop
: Lists persons using RDMS.

whod
: Lists persons currently using an RDMS data base.

hmp
: Prints the number of people logged in who are using RDMS.

hmd
: Prints the number of people logged in who have a data base set.

10) Command Typing and Control

(See also Section 1, "The Multics Command Language Environment", in PART II - REFERENCE GUIDE TO MULTICS of the MULTICS PROGRAMMERS' MANUAL.)

abbrev — Multics Command allowing abbreviation of frequently typed commands. (MULTICS PROGRAMMERS' MANUAL - Command Section)

sms_abbrev — RDMS program allowing abbreviation while using eds, the relational editor. See also the discussion of eds abbreviations in the writeup on eds in this manual.

exec_com — Multics command allowing one command line to invoke a sequence of command lines. (MULTICS PROGRAMMERS' MANUAL - Command Section)

answer — Provide an answer to an expected question during one command line. (MULTICS PROGRAMMERS' MANUAL - Command Section)

start,
release,
program_interrupt — All these are related to stopping and possibly restarting a program after it has started running, and are discussed in the MULTICS PROGRAMMERS' MANUAL, Section on Commands, under the individual descriptions for the commands start, program_interrupt, and release, respectively.

login,
logout — The protocol for logging in is discussed in the REFERENCE GUIDE TO MULTICS, PART II of the MULTICS PROGRAMMERS' MANUAL, Section 1, subsection 1.2. Logging out is also discussed. The logout command in the Command Section of the MPM will also be of interest.

11) Inserting Data into a Data Type or Removing Unused Data (Garbage Collection).

eds — Adds data element to data type while adding a row containing the data element to a relation.

restructure_data_type — Performs garbage collection in a data type, updating relations which use the data type to keep them consistent. May convert a data type to a data strategy module different from its origional dsm at the same time. Data elements of table data types may be selectively inserted into, deleted from, combined with, or moved to other positions in the table.

(END)

An Example Data Base
09/24/74

The following is a description of a data base. It describes data bases used by the Departmental Information System's application of the Relational Data Management System facilities. The relations and data types described below will be used as examples throughout this guide.

I. The Departmental Data Base

Associated with each department's use of the Departmental Information System is a data base. This data base contains personnel and directory information and information describing the assigned duties, responsibilities, and their corresponding accounting charges for each member of the Department's faculty and staff. (Figure 1 illustrates the types of data contained within the data base.)

Information from the data base is used in the generation of a number of reports for departmental and school use. These reports assist the individual departments in allocating, and therefore, in effectively utilizing, the resources at their disposal. If the reports are to be useful, the data upon which they are based must be accurate, complete, and current. This is the responsibility of each Department's administrative staff.

This part of the Reference Guide discusses the structure and contents of a DIS data base in order to provide the background necessary for those persons having responsibility for creating and maintaining data bases using the relational data management facility.

A DIS data base consists of four relations containing information about the departmental faculty and staff. A relation is a file stored within the computer containing information used in the generation of departmental and school reports. In particular, the concept of a relation speaks of data association; that is, elemental pieces of data in a relation are associated in a systematic way one with another. Relations are characterized by the following properties:

a. Independent of how a relation is stored and manipulated within the computer, it can always be thought of as a table containing rows and columns. A familiar example of a relation is a telephone directory. There each entry (row) consists of three distinct elements: a name, an address, and a telephone number.

b. Associated with each column of the relation is an identifier called a data-type. In the telephone directory, the data-types are "name", "address", and "telephone number".

c. Each row of a relation is called a tuple. For example, a typical tuple from a telephone directory is:

Doe, John 14 May St. 363-4420

Each tuple from the telephone directory contains three distinct elements. The number of elements contained in a tuple is called the order or arity of the relation. (All lines (rows) of a relation have the same number of distinct elements.) Each distinct element of a tuple contains one data-element or datum which is a member of the data-type identifying the column in which that element is located. Thus, in the tuple displayed above,

Doe, John

is the data-element (datum) associated with the data-type "name":

14 May St.

is the data-element associated with the data-type "address"; and

363-4420

is the datum associated with the data-type "telephone number". Within a relation a tuple indicates that all data-elements of the row when taken together form a meaningful piece of related information.

d. Every relation is sorted according to some precedence of data-types. The precedence of data-types is indicated by a reading of the data-type names from left to right. A relation may be sorted in many different orders, but the sort order which is judged most convenient (for example, with the telephone directory -- name, address, telephone number) is called the preferred sort. For convience, the list of data-types giving their precedence in terms of sorting is called the sort order of the relation, or (for historical reasons) the ent of the relation.

The relations utilized by the Departmental Information System are described in the following paragraphs.

1. Relation Name: phone_book

The relation named "phone_book" is similar to the Institute telephone directory and contains the data-types "name", "room", and "extn". A brief example of "phone_book" is shown below as it would appear when printed by the command "print_set". (The command "print_set" as well as other commands used in the normal management of a data base are described elsewhere in this guide.) The vertical bar indicates the division of each row into three distinct elements, as specified by the data-types.

sort order: name room extn

Canning, H. F.	36-301	33631
Gilbert, R. G.	36-317	33637
Lambert, P. V.	36-303	33633
Sudbury, A. D.	39-214	34712
Sudbury, A. D.	39-214	34720
Sudbury, A. D.	39-325	33572

Notice that the relation is sorted by name, and that names are stored last name first to permit alphabetical sorting. If a person has more than one "room" or "extn" then that person will have multiple lines (rows) in the relation, one for each triple of information. These multiple lines will be sorted, first by "room" and then by "extn" if a "room" has more than one telephone extension.

All elements of a line need not be present. For example, If no telephone extension were associated with P. V. Lambert, then the row of the relation containing Lambert's name would appear as:

Lambert, P. V.	36-303	

This situation is described by saying P. V. Lambert has a null "extn".

2. Relation Name: rank_list

The "rank_list" relation consists of information concerning appointments for the current fiscal year. Its data-types in the preferred sort order are:

| name | rank | title | soc_sec_num |

The portion of the line "soc_sec_num" refers to the social security number of the individual named in the "name" column of the tuple. A sample rank_list is printed below (again in "print_set" format).

sort order: name rank title soc_sec_num

Canning, H. F.	PROFESSORS	Jones Professor	071423837
Gilbert, R. G.	ASSISTANT PROFESSORS		349824917
Lambert, P. V.	LECTURERS		017849238
Sudbury, A. D.	PROFESSORS		8340120034

Typically, "title" is assigned the null value if it can be derived from the rank and department name. Thus, an Associate Professor in the Department of Civil Engineering whose title is "Associate Professor of Civil Engineering" would typically have a null title.

The data-type "rank" is a table data-type and thus data-elements are restricted to the following list:

"DEPARTMENT HEAD"
"DIRECTOR"
"PROFESSORS"
"ASSOCIATE PROFESSORS"
"ASSISTANT PROFESSORS"
"ADMINISTRATION"
"LECTURERS"
"INSTRUCTORS"
"RESEARCH ASSOCIATES"
"RESEARCH STAFF"
"DSR STAFF"
"TEACHING ASSISTANTS"
"RESEARCH ASSISTANTS"
"EXEMPT PERSONNEL"
"SECRETARIES"
"TECHNICAL SUPPORT STAFF"

The "rank_list" relation is the key relation for many of the reports generated by the Departmental Information System. Any individual who does not appear in the "rank_list" relation will not appear in any of these reports. Thus, it is particularly important that this relation be carefully maintained. When performing a large number of updates to the data base, all updates to the "rank_list" relation should be performed first.

3. Relation Name: sal_list

The relation "sal_list" contains current salary information for each member of a department's faculty and staff. Its preferred sort order and data types are:

| name | term | percent_time | sal_type | salary | hours |

"full_time_equivalent" is understood to be the number of full time equivalents represented by this individual's total activity for this "term" in this department. It is expressed as a percentage. If "full_time_equivalent" is entered as null, "100" percent is assumed. "salary" is the individual's salary for a pay period or pay unit. A person's pay period is specified by his "salary_type" and if "salary_type" is "hourly" then the person's "hours", the number of hours in a pay period must also be entered. The data types "term" and "salary_type" are restricted to the following tables of values:

term

"SUMMER SESSION"
"FIRST TERM"
"SECOND TERM"

salary_type

annual
map
summer
special
monthly
bi-weekly
hourly

A sample listing of "sal_list" follows, again printed in the output format used by the "print_set" command.

sort order: name term percent_time sal_type salary hours

Canning, H. F.	SUMMER SESSION		summer	5000.00	
Canning, H. F.	FIRST TERM	50	map	11250.00	
Canning, H. F.	SECOND TERM		map	11250.00	
Gilbert, R. G.	SUMMER SESSION		summer	2800.00	
Gilbert, R. G.	FIRST TERM		map	6300.00	
Gilbert, R. G.	SECOND TERM	50	map	6300.00	
Lambert, P. V.	SUMMER SESSION		bi-weekly	220.00	
Lambert, P. V.	FIRST TERM		bi-weekly	220.00	

Lambert, P. V.	SECOND TERM		bi-weekly	220.00		
Sudbury, R. W.	SUMMER SESSION		summer			
Sudbury, R. W.	FIRST TERM		map			
Sudbury, R. W.	SECOND TERM		map	9000.00		

As an additional protection feature, the salary data is stored in an enciphered form to prevent its being accessed by other than authorized department staff. Since the Multics system already provides protection against unauthorized access to user's data, the enciphering of salary data is designed to allow some but not all users of a particular department's data base access to the salary information.

4. Relation_Name: acad_assign

Of the relations currently utilized by the Departmental Information System, the most important for viable day-to-day operation is "acad_assign". This relation consists of the information which describes in detail the academic, research, and administrative activies of each member of the Department's faculty and staff. Each tuple of the relation represents one assignment. The preferred sort order for the data-types contained within this relation and their names are:

"name"	(as in "phone_book" and other relations)
"term"	(as in "sal_list")
"acct_num"	(M.I.T. account number)
"acct_type"	(account type)
"percent"	(percentage of accountable time spent on this particular assignment)
"role"	(description of activity, e.g., lecture, seminar)
"subj_num"	(subject number, i. e., 1.18, 16.94J, 2.87T, 14.00X)
"date_start"	(date when assignment begins - input only when date begining does not coincide with beginning of term indicated by term)
"date_end"	(date when assignment ends - input only when date ending does not coincide with ending of "term")
"svsr"	(supervisor for this assignment; must be a name that appears in the "name" column of "rank_list")
"comments"	(other pertinent information about this assignment)

It should be noted that the data-types "role" and "acct_type" (and "term" as already has been noted) are restricted

as to their data-elements. That is, these data types are table data-types and their elements are limited to the following lists:

acct_type

"GEN"
"FUND"
"DSR"

role

In-charge
Faculty Counselor
Leave of Absence
Institute Committees
Department Committees
Administration
lectures
recitation
seminar
tutorials
secretary
Graduate Thesis Supervision
Research

In practice, no tuple will contain data for every one of the eleven data-types. Table 1 presents an example of the relation "acad_assign" as printed by the command print_set.

While there is some duplication of data in succeeding tuples of of "acad_assign", the editing program "eds" permits the user to omit entering a substantial part of this repeated data (set the "" and "D" requests in "eds"). Also, common data-elements (such as "FIRST TERM"), can be abbreviated on input to shorten the time and effort required in the editing process (see the "~" request in "eds" and the command "sms_abbrev").

II. The Structure of a Data Base

The Departmental Information System uses the Relational Data Management System as the basis for its information storage and manipulation. Thus a department's data base is just a particular application of RDMS and the structure of a DIS data base is just a particular structure chosen by the designers of DIS for its needs. This section uses the DIS data base to explain some of the basic concepts of a RDMS data base.

All of the information in a data base is maintained in a private data-base-directory which is a Multics directory. Access to this data-base-directory and its contents depends upon the access granted to users of Multics by the data-base-directory creator. Since the RDMS system programs must be initialized prior to their use, many users of RDMS automatically issue a "set_data_base" command during their login (or start_up exec_com). The RDMS system only allows one data-base to be initiated at any given moment. However, by using the set_data_base command, a user can maintain data in a number of data bases. Since it is possible to share data between data bases, using different data bases permits use of the Multics access control facilities to control access to certain categories of information.

In addition to the relations (such as the ones described above), two other types of files are contained i a data base and listed in the data-base-directory: data-type-files and system-required-files

System required files are used by the RDMS primitives and normally are of no concern to the user. These files can be identified by their first four characters, namely, "SMS_". The system-required-files include:

SMS_segments (used to record a list of the user's data-type-files and relations)

SMS_dummy_area (used by virtual data strategy modules)

Data-type-files are a division of all data in the data base according to a set of functional attributes. All data-types in a data base have two important characteristics:

a. No data-element may appear in a relation unless it appears in a data-type-file. For example, "Sudbury, R. W." could not be listed in the "phone_book" relation unless his name appeared in the data-type-file called "name".

b. Data-types are in reality a mapping between data-elements (like "Sudbury, R. W.") and numbers. Every element of data found in a data-type has a unique reference number (or refno).

To implement the mapping between data-elements and reference numbers, the RDMS system uses programs called data strategy modules. The standard data strategy modules (dsm's) are

automatically provided when a data base is created (refer to "createdb" and "dsm"), but new ones can be added to facilitate special uses. For instance, the standard dsm's are not designed for MIT room numbers, and the mapping the standard dsm provides does not allow the RDMS system routines to correctly sort room numbers. Thus, a special dsm called "dsm_room" has been provided for those users requiring such facilities. The RDMS system associates with each data-type a data strategy module. Whenever a new data type is created, the dsm which will provide the mapping between data-elements and numbers (refnos) must be specified. In some cases, the dsm needs additional information about the data-type in order to perform its function, and in such a case, it queries the creator of the data type for the additional data.

Relations are also stored in individual files in the user's data base directory. The RDMS system uses programs called relation strategy modules (rsms) to manage the information contained in a relation. At present, the only relation strategy used stores the numbers (refnos) associated with the data elements in a table. (Since computers are generally more adept in processing fixed length numbers than variable length character strings, this type of internal representation makes the system faster. Also, the fixed length numbers are usually much shorter than the character strings they represent and thus a significant storage saving can result if the same character string (such as a name) appears in many tuples of one or more relations.) As an example, consider the mechanics of the relation "phone_book". As printed by the "print_set" command, "phone_book" appears as:

```
! Canning, H. F. ! 36-301 ! 3631  !
! Gilbert, R. G. ! 36-317 ! 33637 !
! Lambert, P. V. ! 36-303 ! 33633 !
! Sudbury, A. D. ! 39-214 ! 34712 !
! Sudbury, A. D. ! 39-214 ! 34720 !
! Sudbury, A. D. ! 39-325 ! 33572 !
```

Internal to the computer, however, the relation is represented by a table of numbers which can be printed by the "print_set" command. Using the "-decimal" option to "print_set", the relation is printed with the decimal representation of the numbers assigned to each character string data-element in the relation. The relation "phone_book" would look like:

```
sort order: 81  82  83

 !  8589934592 ! -15032231424 ! 33631 !
 ! 12884901888 ! -15032223232 ! 33637 !
```

21474836480	-15032230400	33633
27380416512	-13421663232	34712
27380416512	-13421663232	34720

The data strategy module for a particular data-type-file using the information contained in the data-type-file maps these reference numbers into their data-element values (or vice versa). The mapping associated with the data-type for "name" is:

refno	character string (datum)
8589934592	Canning, H. F.
12884901888	Gilbert, R.G.
21474836480	Lambert, P. V.
27380416512	Sudbury, A. D.

The mapping for "room" is:

-15032231424	36-301
-15032223232	36-303
-15032230400	36-306
-13421663232	39-214

And the mapping for "extn" as assigned by the data strategy module "dsm_integer" is:

33631	33631
33637	33637
33633	33633
34712	34712
34720	34720

Notice that while each data strategy module uses its own method of mapping character strings into reference numbers and vice versa, the sorting order of the data-elements, be it alphabetical for names or numerical for extns, and the sorting order of the reference numbers (which is numerical) are one and the same. This use of reference numbers is exceedingly useful in providing fast and efficent information management facilities.

While the use of reference numbers plays a major role in the internal workings of RDMS, the non-programing user will not have to manipulate them directly. All of the interfaces for the non-programmer to the RDMS facilities deal with the character string representation of the data and the names of data-type-files and relations.

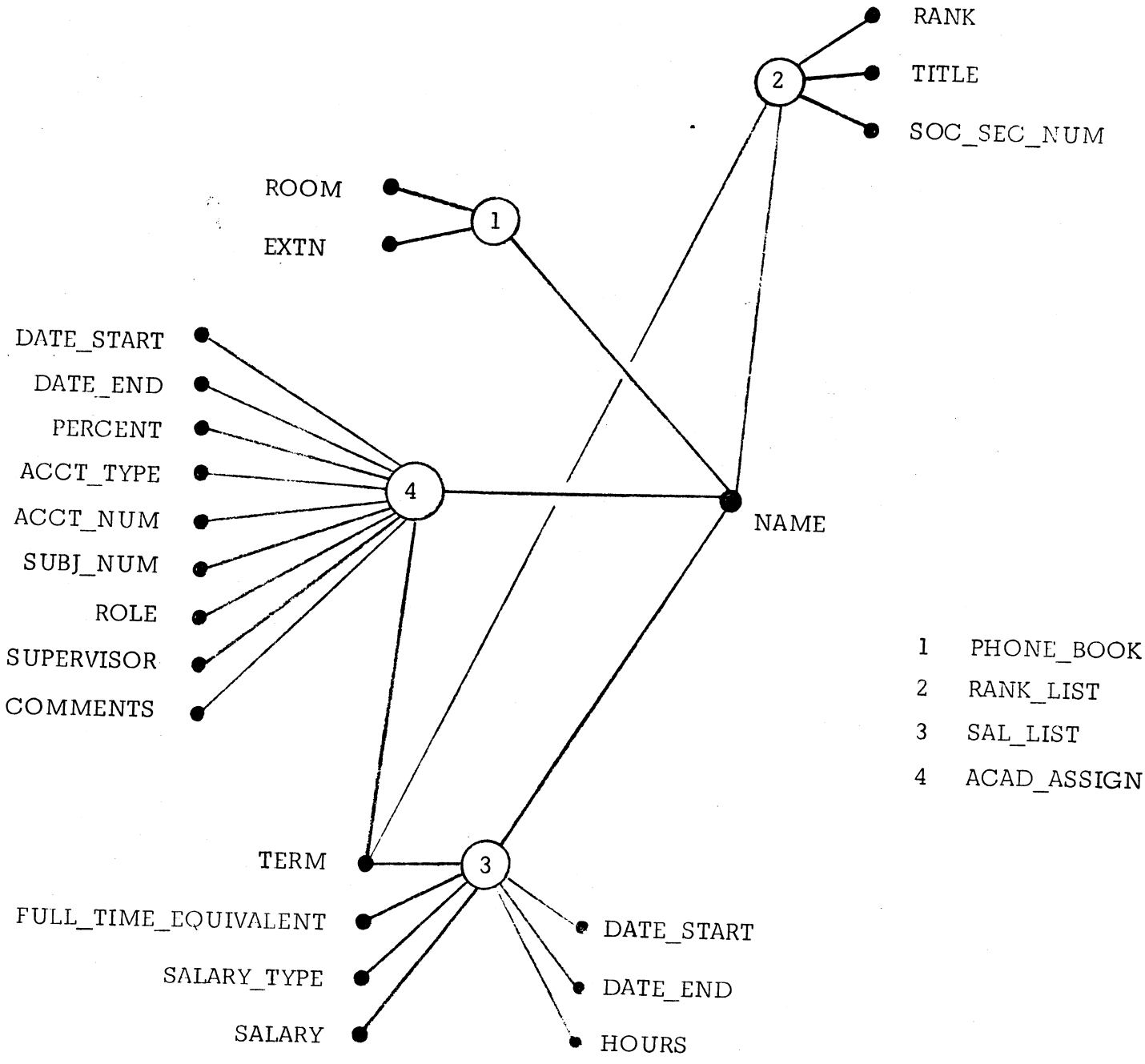


FIGURE 1

A Pictoral Representation of the Standard School of Engineering Departmental Information Data Base

sort order: name term role acct_num acct_type percent subj_num date_start date_end svsr comments

Canning, H. F.	SUMMER SESSION		71234	DSR	100		June 1, 1971	July 31, 1971		
Canning, H. F.	FIRST TERM	in-charge			6.16				
Canning, H. F.	FIRST TERM	in-charge			6.17				
Canning, H. F.	FIRST TERM	Faculty Counselor	10401	GEN	5				Year 2
Canning, H. F.	FIRST TERM	Institute Committees	10405	GEN	10				
Canning, H. F.	FIRST TERM	*IC	10405	GEN					Institute Committee on Curricula
Canning, H. F.	FIRST TERM	*IC	10405	GEN					Institute Committee on Resources
Canning, H. F.	FIRST TERM	Department Committees	14060	GEN	5				
Canning, H. F.	FIRST TERM	*DC							Department Committee on Graduate Engineering
Canning, H. F.	FIRST TERM	recitation	10401	GEN	40	6.16			(two sections)
Canning, H. F.	FIRST TERM	Research	71234	DSR	20				
Canning, H. F.	FIRST TERM	Research	71235	DSR	20				
Canning, H. F.	SECOND TERM	in-charge			6.16				
Canning, H. F.	SECOND TERM	in-charge			6.161				
Canning, H. F.	SECOND TERM	in-charge			6.17				
Canning, H. F.	SECOND TERM	Faculty Counselor	10401	GEN	5				Year 2
Canning, H. F.	SECOND TERM	Institute Committees	10405	GEN	10				
Canning, H. F.	SECOND TERM	*IC							Institute Committee on Curricula
Canning, H. F.	SECOND TERM	*IC	10405	GEN					Institute Committee on Resources
Canning, H. F.	SECOND TERM	Department Committees	14060	GEN	5				
Canning, H. F.	SECOND TERM	*DC							Department Committee on Graduate Engineering
Canning, H. F.	SECOND TERM	recitation	10401	GEN	35	6.17			
Canning, H. F.	SECOND TERM	laboratory	10402	GEN	15	6.161			
Canning, H. F.	SECOND TERM	Research	71234	DSR	30				
Gilbert, R. G.	SUMMER SESSION	Faculty Counselor	14401	GEN	10				
Gilbert, R. G.	SUMMER SESSION	Research	71236	DSR	90		July 1, 1971	August 31, 1971	
Gilbert, R. G.	FIRST TERM	Leave of Absence							
Gilbert, R. G.	SECOND TERM	recitation	10401	GEN	40	6.16			
Gilbert, R. G.	SECOND TERM	laboratory	10402	GEN	20	6.161			
Gilbert, R. G.	SECOND TERM	Research	10451	GEN	40				
Lambert, P. V.	SUMMER SESSION	secretary	71234	DSR	100				
Lambert, P. V.	FIRST TERM	secretary	10401	GEN	20	6.16			
Lambert, P. V.	FIRST TERM	secretary	10402	GEN	20	6.161			
Lambert, P. V.	FIRST TERM	secretary	10405	GEN	10				for Committee Members
Lambert, P. V.	FIRST TERM	secretary	71234	DSR	20				
Lambert, P. V.	FIRST TERM	secretary	71235	DSR	30				
Lambert, P. V.	SECOND TERM	secretary	10401	GEN	20	6.16			
Lambert, P. V.	SECOND TERM	secretary	10402	GEN	20	6.161			
Lambert, P. V.	SECOND TERM	secretary	36306	FUND	20	6.17			
Lambert, P. V.	SECOND TERM	secretary	71236	DSR	40				
Sudbury, R. W.	SUMMER SESSION								
Sudbury, R. W.	FIRST TERM	in-charge			6.188				
Sudbury, R. W.	FIRST TERM	in-charge			6.189				
Sudbury, R. W.	FIRST TERM	recitation	10402	GEN	40	6.188			
Sudbury, R. W.	FIRST TERM	Research	71234	DSR	60		September 1, 1971	October 31, 1971		
Sudbury, R. W.	FIRST TERM	Research	71235	DSR	30		November 1, 1971	January 15, 1972		
Sudbury, R. W.	FIRST TERM	Research	71236	DSR	30		November 1, 1971	January 15, 1972		
Sudbury, R. W.	SECOND TERM	lectures	36306	FUND	40	6.189				
Sudbury, R. W.	SECOND TERM	Research	71236	DSR	60					

(END)

TABLE 1

An Example of the Relation 'acad_assign'

INDEX

This index covers Part I (Introduction) and Part II (Commands) of the RDMS REFERENCE GUIDE.

Following each topic there can be up to three types of references: 1) RDMS documents are named first. These are indicated by the name of a command write up ("add_name_set") or a section of Part I of the RDMS REFERENCE GUIDE ("1.1 Introduction"). 2) Documentation of interest that may be found in the Multics Programmers' Manual (MPM) appears next. These entries are preceeded by "(M)" to distinguish them from refererences to RDMS documents. The MPM is divided into two parts: Part I is the "Introduction to Multics" (abbreviated MPM I), and Part II is the "Reference Guide to Multics". Each part is divided into chapters or sections ("MPM II, 1.4" is the abbreviation for Part II of the MPM, Section 1 on "The Multics Command Language and Environment", subsection 4 on "The Command Language".) In addition, Section 9, "Commands", of Part II contains write-ups of Multics commands. These are indicated by name ("exec_com"). 3) Finally, one topic may refer the user to another topic. For example, under the topic "cost saving features" appears an entry "(see absentee usage)".

& convention

* convention

<

(M) MPM II, 1.5

= convention

(M) MPM II, 1.5

>

(M) MPM II, 1.5

Daemon

(M) dprint

EOR

eds

I/O

(M) MPM II, 4

Job Control Language

(M) exec_com
(see absentee usage)

absentee reports

(see absentee usage)

absentee usage

hmd
hmp
whod
whop
(M) cancel_abs_request
(M) enter_abs_request
(M) exec_com
(M) hmu
(M) list_absentee_request
(M) who
(see active functions)

canonicalization

(M) MPM II, 1.3

cardinality

1.1.Introduction
1.6.Design Principles
display_relation
print_set
status_set

catalogs

(M) MPM II, 3
(see data base)

changing names

add_name_set
change_name_set
(M) MPM I, 1.12
(M) addname
(M) deletename
(M) fs_chname
(M) rename

cleanup tools

delete_set

collating sequence

(M) sort_file

command language

(M) MPM II, 1

creating relations

copy_set
create_relation
eds
sms_interface

creator

(M) status

data base

1.1.Introduction
1.3.Example Data Base
1.6.Design Principles
createdb.ec
set_data_base

data format

2.1.DSM Overview
2.2.DFM Overview

data representation

(see data format)

data strategies

2.1.DSM Overview

data type

1.1.Introduction
2.1.DSM Overview

datum

(see data type)

exec com

(M) exec_com
(see absentee usage)
(see active functions)

formating

(see data formats)

formats

(see data formats)

handling unusual occurances

(see error handling)

help

(M) help
(see check_info_segs)

information

(see help)

initiating a data base

set_data_base

initiation

(M) initiate

input

eds

Input into a relation

eds

Inserting data

eds
restructure_data_type

Inserting data strategy modules

(see inserting modules)

Inserting data types

copy_set
insert_module
insert_set
new_data_type

Inserting modules

insert_module

Inserting relation strategy modules

(see inserting modules)

Inserting relations

copy_set
insert_set
(see inserting modules)

Inserting tuples into a relation

eds

Inter-user communication

(M) mail
(M) send_message

messages

(M) send_message

modifying relations

eds
restructure_data_type
sms_interface

moving information

sms_interface

moving names

add_name_set
change_name_set
rename_set

moving relations

copy_set

name copying

add_name_set
change_name_set
set_name

new relation

create_relation
eds
make_relation
sms_interface

offline

(M) absentee usage
(M) dprint

process interruption

eds
(M) program_interrupt
(M) release
(M) start

project (set theoretic)

sms_interface

protection

eds
(M) MPM II, 3.4

quart

1.1.Introduction
1.6.Design Principles

quasi-relation

(see quart)

quits

(see program interruption)
(M)

quitting

(see program interruption)

ready message

(M) MPM I, 3-5
(M) ready

union (set theoretic)

sms_interface

verbose mode

eds

virtual data type

1.3.Example Data Base
1.6.Design Principles
2.1.DSM Overview

width (of a relation)

(see order)

(END)

This glossary defines several terms which are used throughout the RDMS REFERENCE GUIDE.

data-type
: A collection or set of data elements, such as names, telephone numbers, salaries, or titles. (In other places the term "data-type" refers to objects such as integers, character strings, or complex values. RDMS is more oriented toward a user-view than an implementers view. Writers of compilers deal with integers and strings, users deal with salaries, telephone numbers, titles, and person's names.)

dfm
: short for "data format module". A program which takes a reference number and a character string, and which modifies the character string to have a new format ("Jones, Edward" -> "Edward Jones") or calculates the string from the reference number (refernce_number = 19740801, string output is "August 1, 1974"). See the Overview on DFMS.

dsm
: short for data strategy module. A program which knows everything necessary about one class of data elements. f rdms.if "page2" See the Overview on DSMS.

module
: A program which manages a data-type or a relation. A program which manages a data type is called a Data Strategy Module (dsm), while a program which manages a relation or quart is called a Relational Strategy Module (rsm).

quart
: for "quasi-relation". A quart has most but not all of the characteristics of a relation. The differences are: i) a quart is not stored in the data base directory and will not survive a crash, while a relation or temporary relation is stored in the data base directory and will survive a crash. ii) All of a user's quarts are stored in the same Multics segment, while relations are usually stored in separate segments. Since a segment has a maximum size, the maximum size of a relation may be larger than the maximum size of a quart. iii) Multics access control operates at the segment level. Thus a user may give another user access to a relation, but information in a quart cannot easily be shared. The commands "cleanup_data_base" and "terminate_data_base" delete all quarts.

relation A relation is a set of relationships such as "John has telephone 253-111 and room 39-400", "Mary has telephone 253-2222 and room 39-500". In its narrowest sense, "relation" refers to a non-temporary relation which is a permanent part of a data base. Used more generally, "relation" may refer to a non-temporary relation, a temporary relation or a quart, all of which are collections of relationships. But temporary relations and quarts are not permanent parts of a data base.

rsm for "relational strategy module". An rsm is a program (or module) which knows everything necessary to manage one scheme for representing a relation. Currently, RDMS only uses "rsm_matrix", but other rsm's might be "rsm_multi_segment_file_matrix", "rsm_linked_list" or "rsm_tree".

set A set is a collection of elements. Data types are collections of datums, while relations are collections of relationships. Thus the RDMS command "list_sets" tells a user what relations and data-types are in the user's data base.

temporary relation
A temporary relation is a relation stored in the data base directory, but having a special name such as "+TEMP+.!BBBQdFairGpPXZT". Temporary relations are deleted when a data base is terminated. (See documentation of "terminate_data_base".)

virtual As in "virtual data type". RDMS uses the word to suggest something which has a logical existance but does not physically exist. Thus a data-type named phone_number managed by dsm_integer would be virtual, because one can envision a collection of "phone_number"'s logically, but RDMS does not implement such a collection as a Multics segment containing the integers. By contrast, a data-type named "name", managed by dsm_v2_astring would b non-virtual, because in addition to the logical collection of names, a Multics segment containing a list of names will also exist. No "virtual relations" are currently implemented, so all relations are non-virtual.

(END)

1. Introduction

This section describes the underlying concepts and functional capabilities of an advanced generalized data-base management system based on the set-theoretic concepts of a thesis (8) on the "GOLDSTAR" system written by L. A. Kraning and A. Fillat of M.I.T. in 1970. This system provides a generalized data-base management and reporting facility based on explicit mathematical concepts, but is conceptually simple enough to be readily used without requiring the user to alter his perception of his needs. This system, named Relational Data Management System (RDMS), is currently implemented in PL/1 on the Multics operating system and is being used in many areas to serve administrative needs at M.I.T.

The Relational Data Management System (RDMS) offers a generalized data-base management system with a uniform system for cataloging files (known as relations) and character string data, and a set of commands and subroutines which operate on the relations of a data-base. In addition, RDMS provides an editor for entering and modifying data, commands for printing selected contents of a data-base and generating reports, and a language for report generation and data-base query. There also exists a generalized computational facility to operate on relations of a data-base.

2. Relations, Columns, Rows

A relation may be considered a computer-stored form of a table consisting of columns, each of which has a distinct meaning specific to that column. For example, consider a telephone book. This may be considered a relation of three columns: name, address, and phone number. Each of these columns has its own way of displaying the data in that column. The entries in the "name" column are of the form "Doe, John C.", and the entries of the "address" column are of the form "226 Farm Rd. Lex" where "Lex" is an abbreviated town name. The entries of the "phone number"

column are of the form "253-4107". Each of these three columns has a specific meaning, and because of this, each column has a particular format for displaying and storing its data. In the terminology of the Relational Data Management System, each of these columns of the relation is called a "dataclass" (referred to as "datatype" in older documentation), a statement that each column of a relation has a specific meaning and type of data as its entries. Each column of a relation is labeled by the name of its dataclass. This name is also referred to as the "column name". Each row of the "phone book" relation is an association between an entry of the "name" dataclass, the "address" dataclass, and the "phone number" dataclass. In RDMS terminology, each row of a relation is known as a tuple. The word "tuple" is a mathematical term which derives from our inability to express associations between more than three objects (datums). The terms "pair", and "triple" could apply to rows of relations with two or three columns, but for four or more, this association is known as a "4-tuple" or "6-tuple", etc.. Thus the word "tuple" is taken to be a general term which includes "pair", "triple", and all other associations between a fixed number of objects (datums). The terms "tuple" and "row" are interchangeable.

A file may be a telephone book as discussed above, or a set of index cards, manila folders, or any usable collection of related information. A file is actually a set of records, whether they are medical records, dossiers, transcripts, etc.. See Figure 1 for an example of a file showing a collection of manila folders, each corresponding to a record of the file. Note that "file" and "relation" are logical concepts. The physical implementation of these concepts in a computer system will be discussed below.

A conventional computer file (called a file) is also a collection of records. Since mass storage devices commonly used with computer systems have continuous storage mediums, a file must be subdivided into logical entities called "records" in computer terminology. Furthermore, each record is subdivided into "fields", each of which corresponds to a particular part of the information of the record. Note that the record consists of a set of fields, each of which contains information related to the information in other fields of that record. Refer to Figure 2 for a diagram of a conventional computer file organization. A relation differs from a record oriented tape or disk file in some ways. However, there are several similarities. The concept of a record as a meaningful binding of related information is

preserved. In RDMS, the record corresponds to what is called a "tuple", or row of the relation. Both a relation and a file may be visualized as a sequential list of tuples.

An important difference between a record and a tuple is the actual storage of the data. A record consists of a character data string (the number of characters equals the record length) wherein each meaningful part of the record is called a field defined as an internal subset (a start character and a length) of the record's string of characters. Thus a conventional computer file's data is meaningful only in the light of a set of field definitions for that file. Any program accessing that file must understand those field definitions. In such a system, a table of field definitions must be kept, and each program or report must understand the properties of the fields specific to it. Also, the data used in computation (i.e., matching, selecting, sorting, reports, etc.) is the actual character string which may be of variable length.

A relation may be visualized as a table which consists of tuples (or records), each of which corresponds to a row in the table of a fixed number of columns (fields). Each column contains data of a meaning and type specific to that column. Thus a column corresponds to a field and may contain variable or fixed length datums. However, the relation itself contains the column names (dataclass names) of its columns, and thus contains its own field definitions with respect to access or printing of the data. As shown in Figure 3, each relation of a data-base contains a variable number of rows, each with a fixed (for the relation) number of columns, along with a dataclass definition for each column. A relation's rows are normally sorted by the hierarchical ordering of the columns from left to right.

3. A Data-Base as a Set of Relations

An RDMS data-base is a collection of relations , each of which represents a set of relationships between data elements (datums) from different dataclasses (or fields). The relation is self-defining in the sense that it labels its own fields, and associates a meaning to those labels (dataclass names). A dataclass name by which the relations columns are labeled might be "name", "rank", "title", "phone number", "address", or any similar user-supplied class of datums or objects. Two separate relations of the data-base may have a common column name, and although those relations may be logically distinct relationships, they are implicitly linked because they involve the same objects (datums of the same dataclass). See Figure 4 for a diagram of the database shown in Figure 3, where the circles represent relations and the large dots represent dataclasses. The line connecting a dot to a circle implies that the relation (circle) includes the dataclass corresponding to the dot. Note that the "name" dataclass occurs in three relations, and that the "task" name also occurs in three relations. Two relations that have at least one common dataclass name are said to be implicitly linked.

4. Operations on the Relations of a Data-Base

An RDMS data-base is a collection of relations, many of which may be of the same form in the sense that they have the same column names. Consider two relations: relation1 and relation2, for example, the relations task_list_A and task_list_B of Figure 3. If both relation1 and relation2 have the same column names, then a row of relation1 has the same meaning as a row of relation2. In that sense, it is meaningful to combine or operate on these two relations in a standard set-theoretic way to create a third result relation. Three standard RDMS primitive set-theoretic operations are 1) union, 2) intersection, and 3) difference. The operation "union (relation1, relation2)" creates a result relation which consists of all those rows in either relation1 or relation2. For an example of the union operation see Figure 5. The operation "intersection (relation1, relation2)" creates a result relation which consists of all rows which are in both relation1 and relation2. The operation "difference (relation1, relation2)" creates a result relation which consists of all rows which are in relation1 and are not in relation2. The above standard set-theoretic operations operate

in the normal manner to form the set-theoretic union, intersection, or difference of sets of rows. In this sense, a relation is conceptualized as a set of rows, and meanings specific to given column names are ignored.

The relations of an RDMS data-base may not have identical format as in the previous example but may be linked together by a common column name. By virtue of the fact that two relations have a common column name <dataclass>, a logical relationship is implied between data in these two relations. Both relations may have the character string <datum> in the column <dataclass>. This <datum> may be an entry by which both relation files may be meaningfully accessed or reported-on. Note that in Figure 3 the relations salary_list and task_list_B both contain the datum "Smith, William B." in the "name" column. Thus the set of rows of relation1 with <datum> in that column are said to comprise that "information in relation1 about <datum>". Likewise, the set of rows of relation2 with <datum> in column named <dataclass> comprise the "information in relation2 about <datum>". These two groups of information about <datum> from relation1 and relation2 may be of different forms and meanings, since relation1 may have different column names (and meanings) than relation2 has. The only guaranteed common information between these two sets of rows with <datum>, is that <datum> is in column <dataclass> in both sets of rows.

The relation operation "compose" takes both the "information in relation1 about <datum>" and the "information in relation2 about <datum>" and creates a result relation which combines "all information from relation1 and all information from relation2 about <datum>". The columns of this result are those column names derived from either relation1 or relation2. Thus two relations with a common column name may be combined into one relation, and files based on a common key or column name(s) may be independently maintained and updated and at a later time combined using the "compose" operation. This is a useful feature which enables many implicit logical relationships between relations of a data-base. For an example of the compose operation, see Figure 6.

There are two operations which create a result relation from just one relation. The "sort" operation simply creates a result relation by sorting the rows of relation1 based on a user-specified sequence of column names from relation1. The relation relation1 is effectively sorted on the first column name

(key) from that sequence, then the second, etc. The "project" operation creates a reduced output relation by effectively eliminating a user-specified set of relation1's columns, and "compressing" the result to eliminate duplicate rows with the remaining columns. An example of the sort and project operations is given in Figure 7. Note that the "task" dataclass has a predetermined user-specified sort order which is not an alphanumeric sort of the datums. (The concept of user-specified sort orders is covered later in this paper.)

Consider two relations which have no column names in common. A meaningful operation which combines these relation is the RDMS operation "cartesian_product". This operation creates a result relation which consists of each row of relation1 combined with every row of relation2. This operation results in a relation whose column names are those of both relation1 and relation2. The number of rows of the result relation is the product of the number of rows of relation1 and the number of rows of relation2. In Figure 8, the cartesian product operation is used to associate with all rows of task_list_B the project="B".

5. Other Facilities and Capabilities of RDMS

RDMS facilities include the above relation manipulation operations as well as report generation, information retrieval, and computational packages which greatly enhance the utility of the relational concept of data. A command language exists which performs all of the above operations on relations. A relation editor "eds" (edit set) is used to create relations and input, update, and maintain data in relational form. In addition, several report and information retrieval packages have been developed and are being successfully used at M.I.T.: relational_operators (rop), quick_report, book, and sharri.

The relational_operators package provides a generalized computational facility for deriving useful computed results from data stored in relational form. The relation_operators package allows for the specification of algorithms which operate on the data in the columns of an input relation to yield computed results in another output relation. Data computed from the relational data within a fine division of the relation's rows may be immediately applied to other computations based on coarser divisions of the relation's rows. This provides computations of average values (or other statistical quantities) as well as sums,

sums of products, or any specifiable computation. For example, consider the relation tasks_salaries_by_task which resulted from composing total_task_list and sal_list and then sorting by task. Refer to Figure 9 for the output of a simple computation, which for each specific task, sums over all rows of that task the product of the percent_effort and salary columns.

Quick_report is a program which simply prints a relation in tabular form. It allows the use of headers, permutation of columns, user-supplied formats, and the ability to ignore certain columns in printing. Quick_report involves little effort on the part of the user who designs the report, and provides nicely formated reports based on a single relation.

Book is a package which allows, in addition to all of the considerations discussed above, a means of printing relations while making formating and printing decisions based on the semantics of stepping through relations (which columns change, the leftmost column which changed, etc.). Book creates a report which pictures the data and the meanings inherent in the structure of the relations involved. The user specifies this report in a relation containing commands to the book package.

Sharri is an acronym which stands for: simultaneous hash access to relations report language. This package enables the user to generate a hash-table form (inverted file) of a relation keyed on the datums in a column or set of columns to facilitate random access retrieval, and provides a reporting facility whereby a report may be generated based on data retrieved simultaneously from several relations. Sharri is useful in keyed-query type reports and large reports which form a picture based on the semantics of keyed retrieval from several relations simultaneously. The user must write a report-specification which is then assembled into an efficiently interpreted form for further use.

6. Implementation of Dataclasses

Instead of storing the character string datum in each of the columns (fields) of the row or tuple (record), a reference number or token is stored in the actual relation file. The character

string datum is stored in a dataclass file. Each dataclass file has a name, and is a distinct file where the character string datums specific to that dataclass are stored and referenced by their reference number (refno). See Figure 10 for a diagram explaining this as it applies to the relation "task_list_8", shown previously in Figure 3. An important constraint on the assigning of refnos or tokens to datums of a given dataclass is that the refnos maintain the intended sort order of the datums in the dataclass. In the example the dataclass "name" is a means of assigning reference numbers to and storing character string datums so that the alphabetic sort order of the datums is preserved in the reference numbers. This means of storing character string datums in a binary tree was suggested by William A. Martin in an internal M.I.T. working paper (9). The concept of accessing such datums by reference numbers assigned with the binary tree, and of thus providing a uniform key (the reference number) whereby all datums of the data-base may be accessed, originated with the thesis (8) by L. A. Kraning and A. Fillat.

A relation is thus represented as a simple matrix of refnos, and this constitutes the entire relation structure. All operations on relations (such as sort, compose, etc.) are operations on this simple matrix of numbers. Since a relation is normally a sorted set of rows, the set theoretic operations of union, difference, and intersect are implemented by stepping through two relation files and the computing time is linear with the size of the relations. The sort operation is simply an algorithm for sorting a matrix of reference numbers. At present, the sort algorithm is a generalization of TREESORT3 (3) to sort the rows of an n by m matrix based on a specified sequence of columns. In addition, although a dataclass (e.g., people's names) may appear in many relations of a data-base, only one dataclass file is maintained. The "task" dataclass in the example (Figure 10) is different from the "name" dataclass file in structure. The "task" dataclass is known as a "table" dataclass wherein datums are assigned reference numbers in the order that they are inserted into the dataclass file, which may be prespecified by the user. This is useful when the user requires a report sorted in a particular order which is not necessarily alphanumeric, such as departmental ranks, or job categories.

The different structures of dataclass require different algorithms for accessing the datums, depending on the dataclass file's structure. In order to provide a general logical means of accessing datums given their reference numbers, and of inserting

datums and assigning reference numbers, the concept of "data-strategy-module" was developed. The RDMS system associates with every distinct dataclass (column) name in a data-base the name of the access-method for accessing datums in that dataclass, and assigning refnos, etc. These names associated with every dataclass are called "data-strategy-modules". They are the names of the procedures which manage a given structure of dataclass file. By relegating these concerns to a level invisible to the user, a single logical interface implements the use of reference numbers, and facilitates the means of storing a relation as a simple matrix of reference numbers.

A relation is implemented as a Multics file and simply specifies bindings (rows or tuples) between the (reference numbers of) datums from a set of dataclasses (one per column of the relation). Each relation is a file which is accessed independently of the representation, storage, method of access, or length of the datums of the specific columns of the relation's rows. Because the reference numbers which are proxies for datums preserve their sort order, the information of this ordering is not lost in the relation file. Physically, a relation is simply a header (with the column dataclass names) followed by a matrix (number of rows by number of columns) of one-word reference numbers. Because of the dataclass and reference number tokening of datums, the concept of a maximum field length is not applicable; a datum fetched from a dataclass file by its refno may yield a character string of any length. This is a useful property which enables the user to specify datums which are meaningful and pertinent to his needs. In this sense, variable-length fields are automatically provided as a byproduct of the tokening of datums of a dataclass. However, relation operations require no reference to the dataclass files and operate entirely on a simple matrix of refnos. The individual datums specified in a relation can be modified by using the relation editor "eds". Increasing or decreasing the size of an individual datum merely changes the refno stored in the relation (and may add a new datum to a dataclass file), without changing the size or storage of the relation. In this sense, replacement of individual datums may be done in place, without chaining or fragmenting the relation file.

A dataclass may be (and often is) used in several relations. Thus for two relations with a common column (field) dataclass name, equal reference numbers correspond to the same character string datum, and all items within the dataclass are maintained in a single dataclass file. Thus if a datum (e.g.,

the person's name "R. Smith") occurs in several relations, it would appear only once in the dataclass file. This indirection and tokening of a dataclass offers several advantages over fields in a record oriented environment. First, all reference numbers are of the same length. This allows uniformity and simplicity of handling of relations as a simple matrix of machine words. Second, it allows compactness of storage since most character strings would require more than one word of storage. True, the character strings themselves must be stored once in the dataclass file, but only the reference numbers, and not the character strings, need to be duplicated in several rows of a relation file, or in two relation files with a common dataclass. Third, it allows fast sorting on the columns (fields) of a relation. Sorting is done on reference numbers (refno's), not on fields of characters. Thus the sort routine, as well as all the other relation manipulation primitive operations, are independent of dataclass files or of the representation of data as stored in the dataclass file. Fourth, it allows extremely powerful, efficient operations based on two relations (files), since in two relations with a common column dataclass, matching reference numbers in those columns correspond to the same character string (i.e., the same data). Thus the operations sort, compose, union, intersection, difference, and project are easily programmed and are quite efficient. In conclusion, the primitive relation manipulation operations, which take as input two relations and yield a third output relation, never need to access any of the dataclass files which are implied by those relations. Such access is needed only at report printing time, and thus extensive manipulations of the relationships or bindings (relations) between datums is possible in a representation-independent, efficient manner. An example of the physical form of a relation is shown in Figure 10.

7. Data Strategy and Format

RDMS thus offers a generalized uniform file interface and a scheme of datum tokening which allows efficient operation on relations. When initially input, a data element is immediately assigned a unique reference number which, from then on, is a token or proxy for that datum and is used in all relations with this dataclass as a column's dataclass. This allows efficiency in relation operations. It also provides protection because a distinct dataclass file may be set up as protected, i.e., readable but not writable. This is useful for controlling the

set of possible datums which may be used in a given column (field) of a relation.

The approach to storage or access of the character string datums is also uniform. Each dataclass file has a unique name (the column's dataclass name). Given the reference number of a datum and the dataclass name, the datum is accessed by RDMS in a uniform way: the call "get_datum_given_refno". This call takes as input the reference number of the datum and the name of the dataclass file; and as output returns a pointer to the character string datum (of arbitrary length).

A uniform interface for obtaining reference numbers or tokens is also provided. Given a character string datum and the name of a dataclass file, the reference number of that datum is determined by the call "get_refno_given_datum". This call takes as input a pointer to the datum and the name of the dataclass, and returns as output the reference number of the input datum. If the datum is to be inserted into the dataclass file and assigned a reference number, the above call (with the "insert" option specified) will do that. If the insert option is not specified the "get_refno_given_datum" call will fail if the datum has not been previously assigned a reference number. Thus RDMS provides automatic detection of possibly incorrect input.

The above two calls comprise the basic logical character string storage and access data-facilities of RDMS, regardless of the stored format or access method of the data specific to the data type. The function of handling the storage and access of data is relegated to what are termed "data strategy modules", which are programs which know a particular data-structure and how to access datums and assign refnos based on that structure. When the user initially creates a dataclass file he specifies the name of a standard or user-provided data-strategy module program (dsm) that he assigns to manage that dataclass. The calls "get_refno_given_datum" and "get_datum_given_refno" are a generalized logical interface to the dsm program specific to a given dataclass file. The RDMS data interface translates a call to "get_refno_given_datum" (or "get_datum_given_refno") into a call to the dsm program managing the dataclass specified in the call. This translation is invisible to and of no concern to the user. A special dsm may be written for a specific application with reasonable ease, and without affecting or modifying any of the other software. In the printing of a report based on a relation, several different dsm programs may need to be accessed

In order to print the different columns of the relation, but the printing programs need call only the "get_datum_given_refno" routine. Examples of useful data strategy modules provided with the RDMS system are

dsm_astring (which assigns refnos to character strings based on a binary tree where the refnos preserve their datum's alphabetical order, good for large amounts of character string data to be used in alphanumerically sorted order. Refer to the "name" dataclass of Figure 10.)

dsm_table (puts character string datums into a table and assigns reference numbers in the user-supplied order that the datums are inserted, good for column (field) entries that take on one of a fixed number of distinct values or codes. Refer to the "task" dataclass of Figure 10. Note that the refnos preserve the user-specified sort order, not an alphanumeric sort.)

dsm_ciph_integer (for cryptographically enciphering salaries, etc.)

A virtual dataclass is one in which the data need not be stored; the reference number is the information. Character string datums are algorithmically related to reference numbers and vice-versa. The standard system-provided virtual dsms are

dsm_integer (a virtual dataclass, an integer is encoded simply as the reference number, and a reference number is simply converted to its integer representation. See Figure 10.)

dsm_date (a virtual dataclass which encodes a date as a reference number and converts a reference number into a date.)

In addition to data-strategy modules as a means of representing and storing datums for later display, there is another level of data representation which is known to all RDMS reporting and information retrieval packages. In these packages, the user may specify the name of a "data-format-module" which is used to provide an alternate to the representation of a datum which is returned by the "get_datum_given_refno" primitive. A

data-format module is a simple (e.g., approximately 10 lines of PL/1) program which converts a character string into an alternate representation. For example, a data-format module named "dfm_name" converts a name of the form "Thompson, James P." into a character string of the form "James P. Thompson". This enables the user to keep names sorted by last name first, but allows a more convenient and useful format for display purposes. A wide assortment of data-format modules have been useful at M.I.T.. The use of data-format modules makes it possible to use a small number of data-strategy modules. For example, dsm_integer simply returns the character string representation of an integer. By the use of the data-format module "dfm_dollar", the integer is modified to look like " $ 45 ", or by the use of "dfm_percent", is modified to look like " 45% ".

8. How RDMS Depends on Multics Features

An RDMS data-base may be considered a set of relation files and a set of dataclass files. The relation manipulation operations (sort, compose, union, intersect, difference, etc.) all require that three relation files be accessible to that operation: the two input relation files and the output relation file. For printing of reports based on relations, the call "get_datum_given_refno" is used to convert the reference numbers stored in the relation file into meaningful character string datums suitable for user reading. This requires that all of the dataclass files for printed columns of the relation must be accessible at some time while printing the report.

In a conventional computer system, a program which performs any of the above operations must keep track of where in main memory parts (buffers) of each of the above files reside, and must be able to specify when to request that other needed parts of those files be moved into main memory. This results in considerable complexity and programming overhead. In addition, each dataclass file must be able to grow independently of other dataclass files and at unpredictable intervals whenever a new datum is inserted into that dataclass file and assigned a reference number. In a conventional system this would require that the length of each dataclass file be dynamically updated at unpredictable intervals. Since there are a number of dataclass structures (one per dsm), each specific dsm would need to keep track of all parts of the referenced dataclass file and be able to request further parts of that file as needed. This is a complex requirement which would require each dsm to be aware of global memory management problems, and to be concerned with the current state of other dataclass and relation files being accessed, in order to effectively manage overlays or to re-use available main memory.

In RDMS, such software complexity is obviated by the features of the Multics Virtual Segmented Memory (refer to Figure 11 and the Multics Virtual Memory paper (2)). Once a file is created, it may at any later time be written into or read by any program by hardware pointers as if it were directly addressable in main memory. The program is unaware of and not concerned with whether the physical locations referenced by those pointers are resident in main memory. Such considerations as disk-id, secondary storage address, length, current main storage location, and the like are handled automatically by the Multics Virtual

Segmented Memory hardware and software. Once a pointer is obtained to a file, a program may address any part of that file as if it were all resident in main memory. Multics provides a hardware scheme which automatically converts a logical pointer address (file name, location within a file) into a suitable physical main memory address, or if that part of that file is not resident in main memory, interrupts the program until it has transferred that part (page) of the file into main storage. All programs written for Multics hardware use the logical 2-dimensional pointer address (file name, location within file). This greatly simplifies operations which need to access more than one file, because it allows programs to directly access and distinguish logically distinct and independent files. This feature is called segmentation, or two-dimensional memory (one dimension is file (segment) name, the other is the location within that file). Due to Multics hardware and software features, writing into a file can in no way affect other files. Multics automatically keeps track of all parts of a given file associated with a file name, and by referencing this file name, it is impossible to accidentally write into portions of another file. Because of the addressing requirements of accessing many files (e.g., relation files, dataclass files) simultaneously, only heavily used portions of which need occupy main memory, a virtual memory with paging on demand is a practical necessity.

RDMS depends heavily on the Multics virtual segmented memory for its software simplicity and efficiency. In Multics, logically distinct files (known in Multics terminology as segments) may be dynamically created and modified, and may grow independently. Each file has a distinct name to which it is known to a user. Multics is a time-sharing system which provides the user with immediate on-line interaction with his data and procedure files. Every Multics procedure file (such as a dsm program or the sort routine) is directly executable because it is hardware accessible by pointers as if the file were (and it is, in the virtual memory sense) resident in contiguous main memory, although its parts (pages) may be scattered throughout the physical main memory resource. Thus the hardware paging and segmentation feature automatically provides conventional loading and linking between parts of a single program. Linking between different program files is handled by the Multics dynamic linking facility, which means that a calling program knows another program only by its name, and when first called, this name is converted to a two-dimensional hardware pointer which is then made known to the calling program by storing it at a predetermined location (specified by the calling program) in a known system segment called the "linkage" segment. In RDMS,

pointers to relation and dataclass files are stored in a similar manner. Therefore, the conventional loading and linking processes are eliminated because the segmented virtual memory automatically provides this function. Thus a software system need not be "loaded" all at once as a fixed load module, and the user pays only for those facilities of the system that he uses. Furthermore, because the RDMS software procedures are "execute-only", a single copy is shared by all users, and the user need have private copies only of his own dataclass and relation files.

Files (segments) in Multics are organized into a set of named directories. A directory is itself a file which logically groups a set of files as specified by the user (refer to Figure 12). Any data file may be said to "belong to", or "be part of" one directory. These directories are organized into a tree structure because directory files may also "belong to" or "be part of" a directory. Individual files and directories have access-control attributes, which serve to specify only those users permitted to read or write into that file or directory of files. Because a file may be created or made known to the hardware only through the use of the Multics supervisor, which is itself protected by hardware, the access attributes of a file may not be compromised by any user, and are checked by the hardware on every memory reference to that file. A file's access attributes with respect to a given user may be either "read", "write", or "execute", or some combination of those. A user may set the access attributes of his files with respect to any other user or group of users. In RDMS, this means that certain relation or dataclass files may be restricted for private use (such as salary_list in the example data-base), while others may be allowed to be "read" for public use. Large relation files may be broken up into two relation files (each with fewer columns) one of which may be given public or controlled sharing, while the other relation file may be restricted for private use. For reasons of organization and uniform access, the relation and dataclass files of a data-base are all in a Multics directory which we call the "data base directory". There exists a master table for each data-base which identifies each relation and dataclass file's name and managing strategy module. This table is named "SMS_segments", and is a permanent file resident in the data-base directory. Refer to Figure 13 for a diagram showing how a data-base is represented as a Multics directory of files. In addition to the names and managing strategy modules of relations and dataclasses, SMS_segments also contains names of the strategy module programs, as well as the name of a managing program for SMS_segments itself.

The ability to name the dataclass files and relation files help the user to keep his data-base organized and understandable. SMS_segments may be likened to the master catalog of a tape or disk library in that it is online-accessible and automatically keeps track of file creation and maintenance. The physical keeping of the files is handled automatically by the Multics virtual segmented memory, and is thus of no concern to RDMS.

It is important to distinguish the Multics file (segment) concept and the RDMS relation file concept. For reasons of system overhead and saving the cost of segment creation, several frequently created RDMS files may be packed into one segment. The RDMS database catalog "SMS_segments" keeps track of which segment a given RDMS file is contained in.

Because the pages of a given Multics file (segment) may be scattered throughout main memory, for each segment currently being accessed the Multics hardware uses a page table resident in main memory to convert a location within that segment into a physical main memory address. By packing frequently created RDMS files into one Multics file (segment), fewer page tables need be created or be resident in main memory, and the system overhead associated with creating, maintaining, and deleting these page tables is reduced.

RDMS operations frequently require small one-use relations to be created and used to specify sort orders for the sort or project operations, or for limited selection operations using the decide_over operation. These temporary relations are not considered to be a permanent part of the database, and do not need to be shared; they are incidental to other RDMS operations. In RDMS terminology such relations are called "quarts", or "quasi relations". They have the structure and meaning of a normal relation which is a separate Multics file (segment), and are used equivalently in relation manipulation algorithms (such as sort, compose, project, union, etc.). However, their creation and deletion is handled differently, and they are restricted in that they cannot grow in length as can normal one-per-segment relations and dataclass files. These quarts are stored in a single Multics segment which is automatically deleted when the user's process or terminal session ends, or when the user "terminates" his database, or switches to another of his databases. Quarts are not kept permanently cataloged as part of the database, and cannot be shared with other users. The cost of creating a quart is a small fraction of that of creating a

Multics segment, and the deletion of quarts is handled automatically.

Another form of temporary relation is useful for an RDMS data-base. All normal one-per-segment relations which are results of a relation manipulation command (such as sort, compose, project, etc.) are supplied a name and are cataloged in SMS_segments with this name. This name is either specified by the user when he issues the command, or is assigned a unique default name "+TEMP+..." because the user did not specify a name. All relations of a data-base which have default names are assumed to be temporary and are deleted when the data-base is terminated or when the user issues the command "cleanup_database". This facility relieves the user from worrying about naming, keeping track of, and deleting relations which are intermediate results. Only those relations specifically named by the user are considered permanent relations of a database.

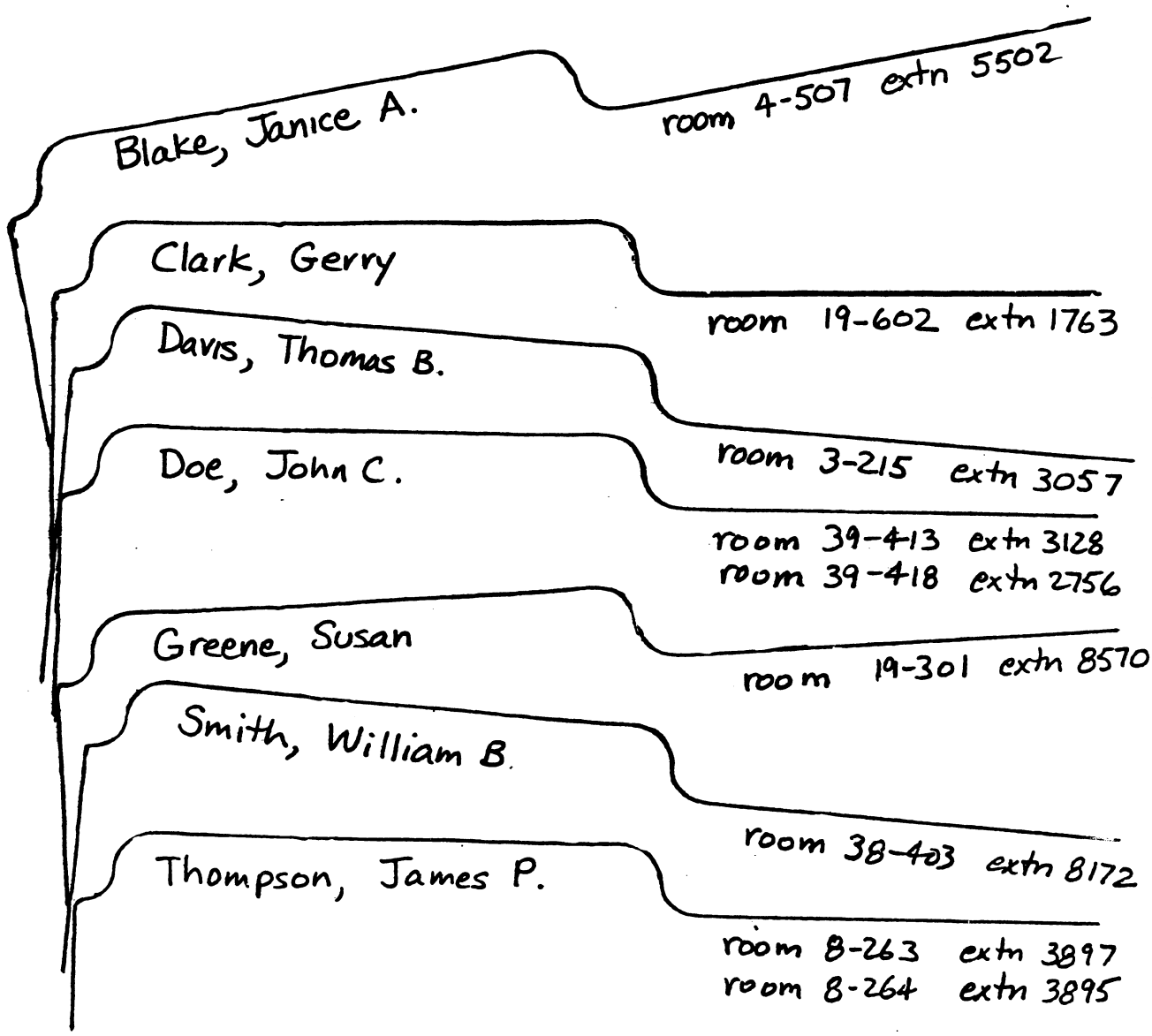
9. Conclusion

The Relational Data Management System evolved from an M.I.T. thesis on the "GOLDSTAR" data management system (8) written by L. A. Kraning and A. Fillat in 1970. The concepts discussed above and the initial implementation originated with that thesis. Since then, the major application of this data management system has been M.I.T.'s Departmental Information System, which is a major administrative tool for departmental administrators in keeping track of personnel, account numbers charged to their roles and efforts, and other complicated reporting functions. The primary user interface to the system is through the relation editor "eds", which evolved through experience into an effective and convenient means of inputting and editing data as relations. The other principle user interface has been through the issuance of commands for generating specific reports which are then picked up at an offline printer. This has involved writing a PL/1 program for each report, which uses the SMS primitive relation commands sort, union, intersect, project, difference, compose, and cart_prod in their subroutine form. A typical report program uses about ten of these operations. Since then many new applications arose, and new facilities and packages have eliminated almost all need for PL/1 coding by users. These packages have proved valuable at M.I.T. in such applications as Financial Planning, Text Information Retrieval, Administration of a Graduate School, and are used in several on-line query systems.

Bibliography

1. Codd, E. F., "A Relational Model of Data for Large Shared Data Bases", Comm. ACM, Vol. 13, No. 6 (June 1970), pp. 337-387
2. Daley, Robert and Dennis, Jack, "Virtual Memory, Processes, and Sharing in MULTICS", CACM, Vol. 11, No. 5, May 1968.
3. Floyd, Robert, "Treesort3", Communications of the A.C.M., August 1964, Computer Associates, Inc., Wakefield, Mass.
4. Goldman, Jay, "Use of Computed Relations in a Set Theoretic Data Base", Unpublished S. M. Thesis, M.I.T., Department of Electrical Engineering, Cambridge, Mass., June, 1973
5. Goldstein, Robert C., and Strnad, Alois J., "The MacAIMS Data Management System", 1970 ACM SIGFIDET Workshop on Data Description and Access. November 1970.
6. Hacker, William Robert, "The Implementation of Data Management Systems Under Multics", unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering, Cambridge, Mass., January 1974.
7. Hansen, Susan Marie, "A Query Language for a Set Theoretic Data Base", Unpublished S. M. Thesis, M.I.T., Department of Electrical Engineering, Cambridge, Mass., January, 1974
8. Kraning, Leslie Alan, and Fillat, Andrew Irwin, "Generalized Organization of Large Data-Bases; A Set-Theoretic Approach to Relations", Unpublished S. M. and S. B. Thesis, M.I.T., Department of Electrical Engineering, Cambridge, Mass., June 1970.
9. Martin, William A., and Ness, D. N., "Optimizing Binary Trees Grown with a Sorting Algorithm", CACM, Vol. 15, No. 2, February 1972.
10. Mason, John T. Jr., "Error Handling in a Set-Theoretic Data Management System", Unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering, Cambridge, Mass., June 1972.
11. Milner, James Michael, "An Implementation of Intercommunication Among Set-Theoretic Data-Bases", Unpublished S. B. Thesis, M.I.T., Department of Electrical Engineering, Cambridge, Mass., June 1972.
12. Multics Programmers' Manual, Honeywell Information Systems Laboratory, 575 Technology Square, Cambridge, Mass., Order Numbers AG90, AG91, AG92, AG93, and AK92.

Figure 1:



The File Concept

Figure 2:

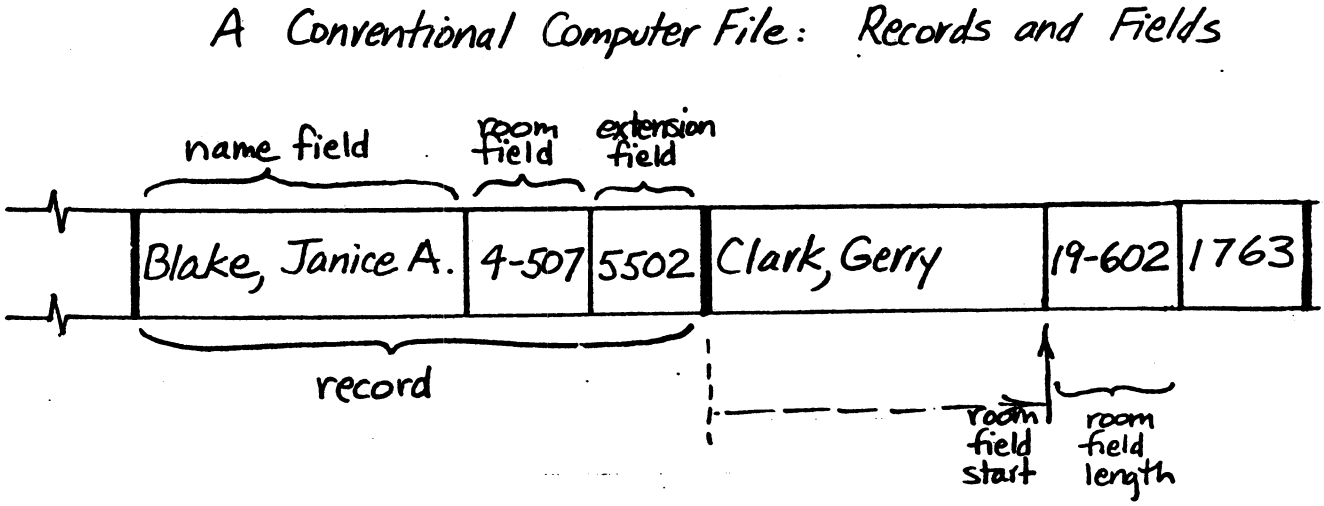

A Conventional Computer File: Records and Fields
name field
room field
extension field
Blake, Janice A.
4-507
5502
Clark, Gerry
19-602
1763
record
room field start
room field length

Figure 3: An RDMS Data-Base

phone_book:

name	room	extension
Blake, Janice A.	4-507	5502
Clark, Gerry	19-602	1763
Davis, Thomas B.	3-215	3057
Doe, John C.	39-413	3128
Doe, John C.	39-418	2756
Greene, Susan	19-301	8570
Smith, William B.	38-403	8172
Thompson, James P.	8-263	3897
Thompson, James P.	8-264	3895

task_list_A:

name	task	percent_effort
Blake, Janice A.	administration	43%
Clark, Gerry	plant supervision	32%
Clark, Gerry	systems design	68%
Doe, John C.	planning	100%
Smith, William B.	implementation	17%
Smith, William B.	scheduling	58%
Smith, William B.	plant operation	25%

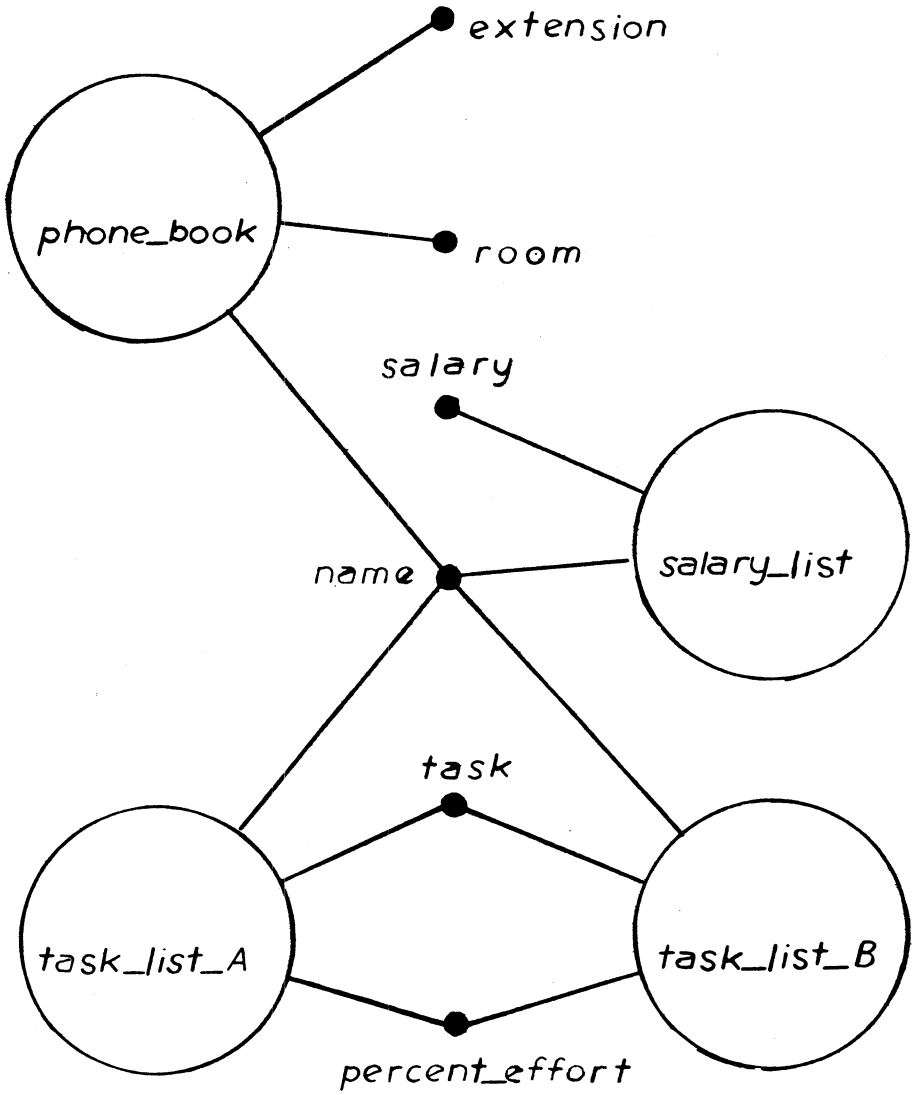
task_list_B:

name	task	percent_effort
Blake, Janice A.	plant operation	57%
Davis, Thomas B.	planning	23%
Davis, Thomas B.	plant operation	61%
Davis, Thomas B.	plant layout	16%
Greene, Susan	plant supervision	100%
Thompson, James P.	administration	100%

salary_list:

name	salary
Blake, Janice A.	$ 23,000
Clark, Gerry	$ 18,000
Davis, Thomas B.	$ 34,000
Doe, John C.	$ 14,000
Greene, Susan	$ 18,000
Smith, William B.	$ 21,400
Thompson, James P.	$ 29,600

Figure 4:



AN RDMS DATABASE:
RELATIONS AND DATACLASSES

Figure 5:

An Example of the Set-Theoretic Union of two Relations

total_task_list = union(task_list_A , task_list_B)

total_task_list:

name	task	percent_effort
Blake, Janice A.	plant operation	57%
Blake, Janice A.	administration	43%
Clark, Gerry	plant supervision	32%
Clark, Gerry	systems design	68%
Davis, Thomas B.	planning	23%
Davis, Thomas B.	plant operation	61%
Davis, Thomas B.	plant layout	16%
Doe, John C.	planning	100%
Greene, Susan	plant supervision	100%
Smith, William B.	implementation	17%
Smith, William B.	scheduling	58%
Smith, William B.	plant operation	25%
Thompson, James P.	administration	100%

Figure 6:

An Example of the Compose Operation

```
tasks_and_salaries = compose( total_task_list , salary_list )
```

tasks_and_salaries:

name	task	percent_effort	salary
Blake, Janice A.	plant operation	57%	$ 23,000
Blake, Janice A.	administration	43%	$ 23,000
Clark, Gerry	plant supervision	32%	$ 18,000
Clark, Gerry	systems design	68%	$ 18,000
Davis, Thomas B.	planning	23%	$ 34,000
Davis, Thomas B.	plant operation	61%	$ 34,000
Davis, Thomas B.	plant layout	16%	$ 34,000
Doe, John C.	planning	100%	$ 14,000
Greene, Susan	plant supervision	100%	$ 18,000
Smith, William B.	implementation	17%	$ 21,400
Smith, William B.	scheduling	58%	$ 21,400
Smith, William B.	plant operation	25%	$ 21,400
Thompson, James P.	administration	100%	$ 29,600

Figure 7:

An Example of the Sort and Project Operations

```
tasks_salaries_by_task =

sort( tasks_and_salaries , by task-name-percent_effort-salary )
```

tasks_salaries_by_task:

task	name	percent_effort	salary
planning	Davis, Thomas B.	23%	$ 34,000
planning	Doe, John C.	100%	$ 14,000
implementation	Smith, William B.	17%	$ 21,400
scheduling	Smith, William B.	58%	$ 21,400
plant operation	Blake, Janice A.	57%	$ 23,000
plant operation	Davis, Thomas B.	61%	$ 34,000
plant operation	Smith, William B.	25%	$ 21,400
plant supervision	Clark, Gerry	32%	$ 18,000
plant supervision	Greene, Susan	100%	$ 18,000
systems design	Clark, Gerry	68%	$ 18,000
administration	Blake, Janice A.	43%	$ 23,000
administration	Thompson, James P.	100%	$ 29,600
plant layout	Davis, Thomas B.	16%	$ 34,000

```
names_task_A = project( task_list_A , by name )
```

names_task_A:

name
Blake, Janice A.
Clark, Gerry
Doe, John C.
Smith, William B.

Figure 8:

An Example of the Cartesian-Product Operation

```
tasks_with_B = cartesian_product( task_list_B , project_B )
```

project_B:

project
B

tasks_with_B:

name	task	percent_effort	project
Blake, Janice A.	plant operation	57%	B
Davis, Thomas B.	planning	23%	B
Davis, Thomas B.	plant operation	61%	B
Davis, Thomas B.	plant layout	16%	B
Greene, Susan	plant supervision	100%	B
Thompson, James P.	administration	100%	B

Figure 9:

An Example of a Relational-Operators Computation
applied to the relation "tasks_salaries_by_task"

```
task_costs = relational_operators( tasks_salaries_by_task , <algorithm> )
```

task_costs:

task	cost
planning	$ 21,800
implementation	$ 3,638
scheduling	$ 12,412
plant operation	$ 39,200
plant supervision	$ 23,760
systems design	$ 12,240
administration	$ 39,490
plant layout	$ 5,440

Figure 10:

THE RELATION-DATACLASS TOKENING METHOD

RELATION: task_list_B
MATRIX OF INTEGER
REFNOS WHICH ARE
TOKENS FOR DATUMS
STORED ELSEWHERE IN
DATACLASS FILES

name	task	percent_effort
2	4	57
5	1	23
5	4	61
5	8	16
8	5	100
12	7	100

DATACLASS: task
DSM: dsm_table

refno=1:	planning
refno=2:	implementation
refno=3:	scheduling
refno=4:	plant operation
refno=5:	plant supervision
refno=6:	systems design
refno=7:	administration
refno=8:	plant layout

DATACLASS: name
DSM: dsm_astring

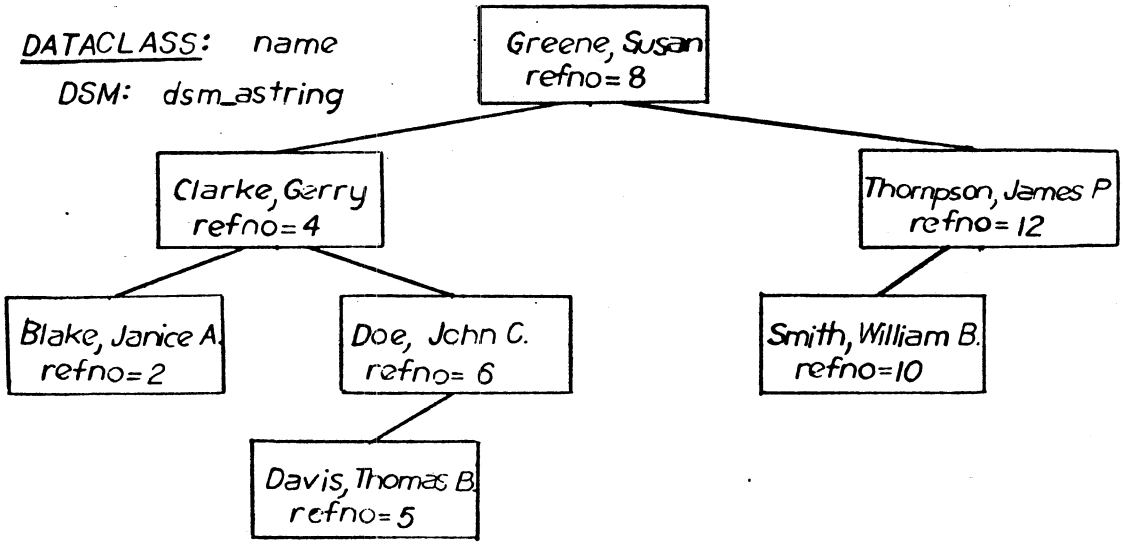


Figure 11:

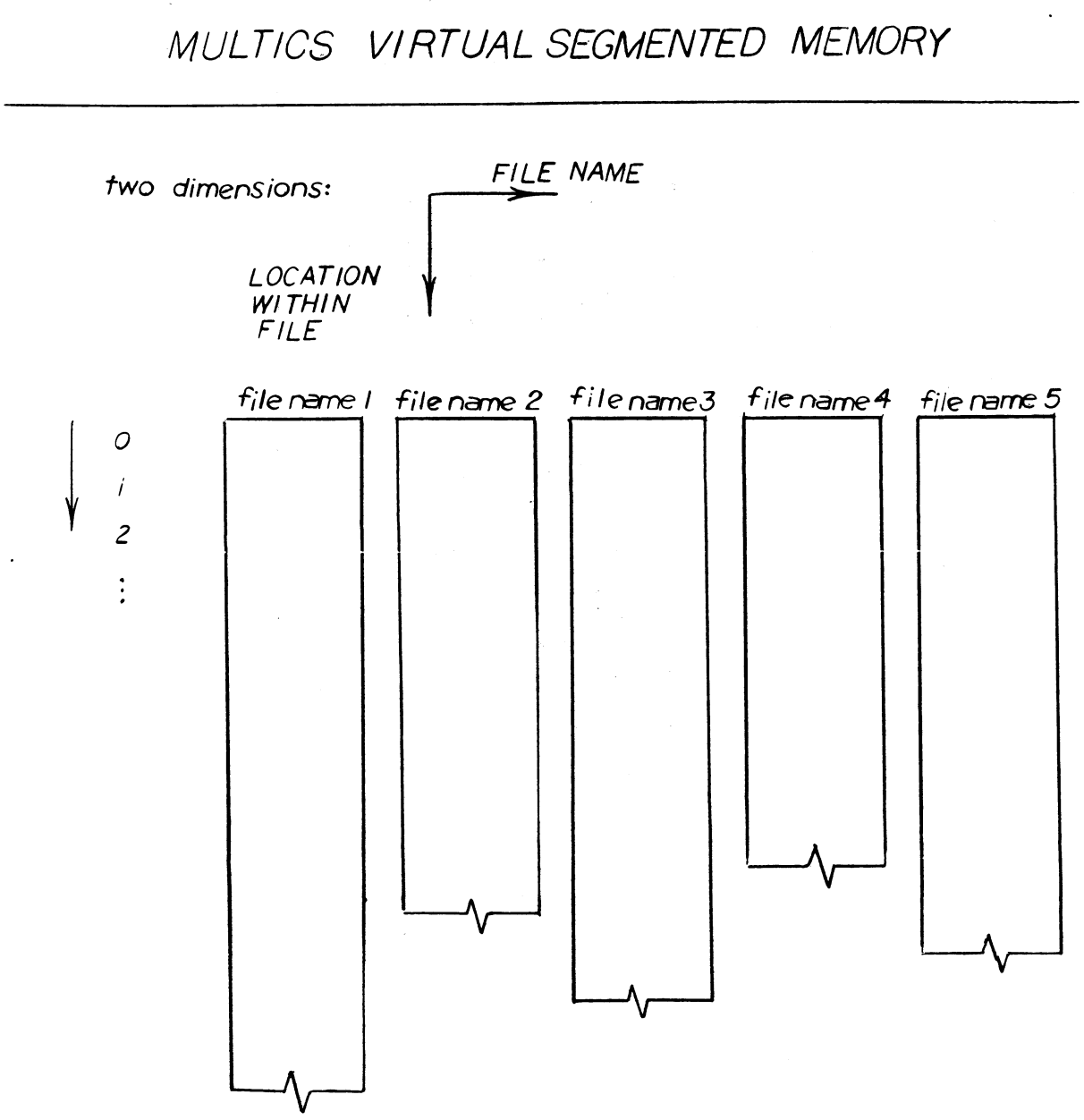

MULTICS VIRTUAL SEGMENTED MEMORY
two dimensions:
FILE NAME
LOCATION WITHIN FILE
file name 1
file name 2
file name3
file name4
file name 5
0
1
2

Figure 12:

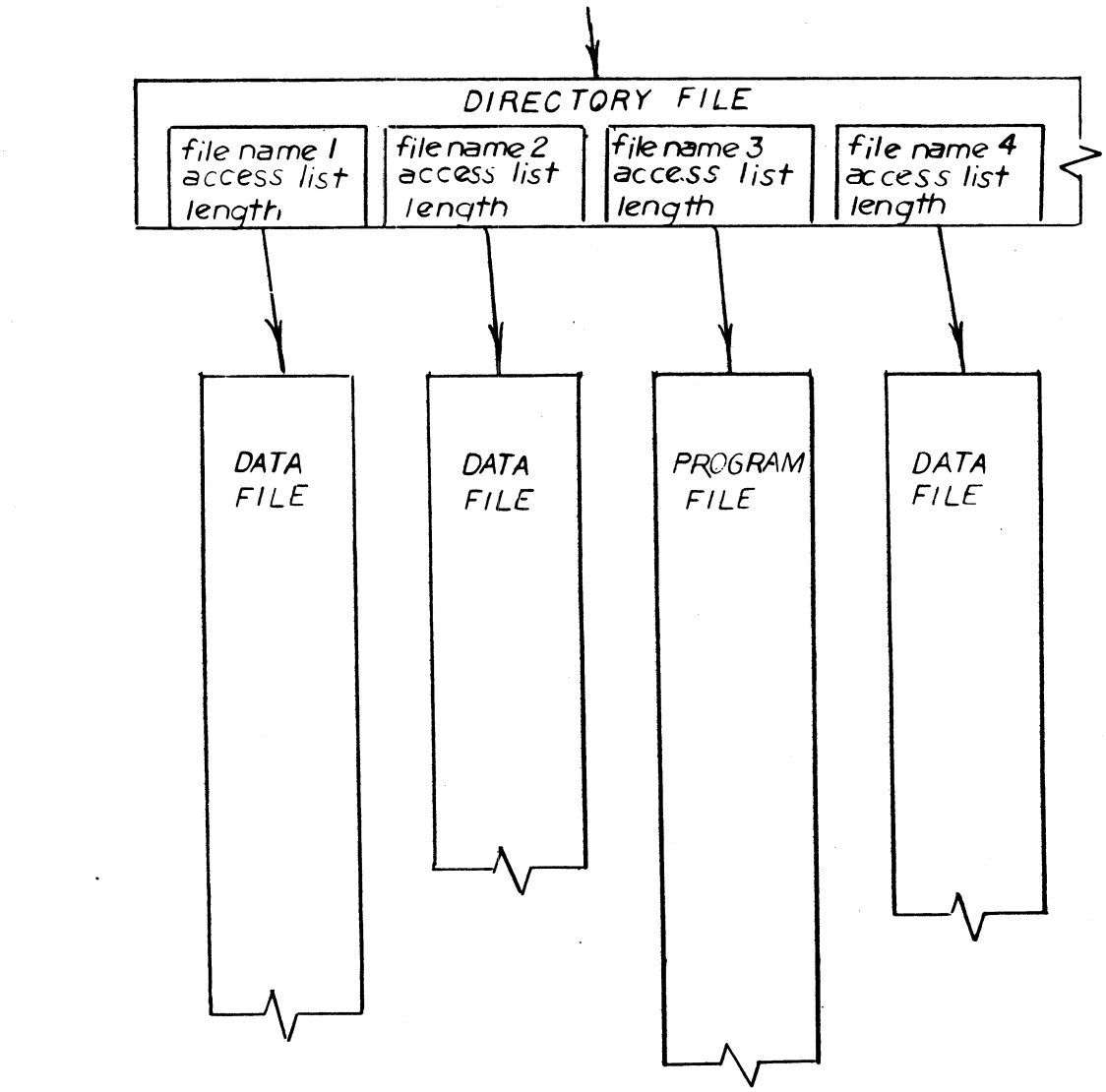
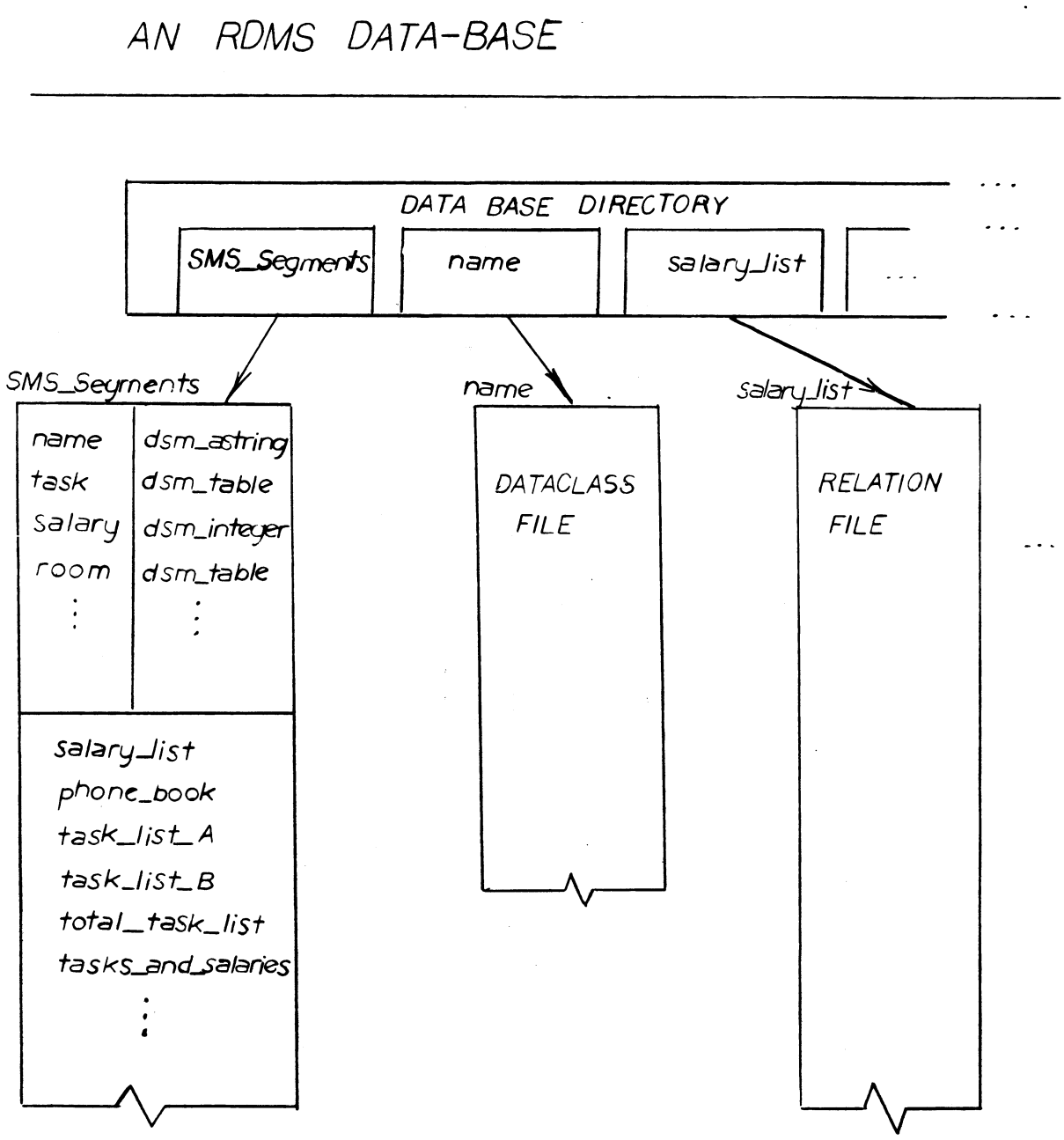

A MULTICS DIRECTORY OF FILES
DIRECTORY FILE
file name 1 access list length
file name 2 access list length
file name 3 access list length
file name 4 access list length
DATA FILE
DATA FILE
PROGRAM FILE
DATA FILE

Figure 13:



(END)

An Overview of Data Strategy Modules
09/24/74

A data strategy module is a program which knows everything necessary about data elements of a particular type. The dsm can be used to manage RDMS "data types", which can be viewed as collections of data elements of that type. The dsm can map a character string representing a data element into a reference number (refno) or token. (The character strings are generally of various sizes while a reference number is always the same size. This makes it much easier and faster to use for such things as sorting, comparison, or copying.) The dsm assigns reference numbers which maintain the sort order desired. The dsm is invoked when inserting a data element into a relation (given the datum it returns the reference number) and when the relation must be printed (given the reference number the datum is returned). All other use of the datum can be by reference number. (Note: the datum is not actually inserted into the relation file. Instead it is first inserted into the data type file and assigned a reference number. Then the reference number is put into the relation to represent the datum.)

Several standard data strategy modules are provided by RDMS. It is also possible for a user to add his own dsm for a particular application.

A user need be concearned with dsm's only when creating a new data type. A data strategy module to manage the data type must be specified. (See the "new_data_type" command.)

The choice of what dsm to use to manage a new data type can be made on the basis of 1) efficiency, 2) need of verification, 3) sort order and 4) convenience. (Some very special purpose dsms may be outside these categories.) The following examples illustrate how these categories relate to common situations.

Suppose a user needs to choose a dsm to manage a data type to be named "phone_numbers", which will include data elements such as "3-4107", "253-7749" and "(617) 253-4605". The user might choose dsm_v2_astring which accepts any character string, but does not verify that a datum as typed by the user is really a telephone number. (If "253-4107" were mis-typed as "253-4107", with the letter "l" substituted for the number "1", dsm_v2_astring would accept both, but would treat them as distinct data elements.) However the user might choose to use dsm_integer which is more efficient than dsm_v2_astring: it translates a character string representing an integer, such as "34107", into the reference number having the same value (34107). dsm_v2_astring has to store the association between "34107" and the reference number assigned to it, which will probably not be

34107. The process of looking up this reference number takes longer. Additionally dsm_integer verifies that it is given a character string representing an integer, so it would reject "2534107" because the letter "l" is non-numeric. In addition, the sort order of integers ("1","2", . . . "10", "20", . . . "100", "200") may be more useful than the sort order of character strings ("1","10","100","2","20", "200"). At the same time, using dsm_integer to manage the "phone_numbers" data type may be less convenient: to input a phone number the user must type "34107" not "3-4107", and the string printed on a report would be "34107". (However a data format module (dfm) could be used so that "34107" would be re-formated as "3-4107" for printing on reports. See the overview of data format modules.) If the user wished to allow a telephone number to include a local extension (such as "(512) 694-3360 extn 523") this could not be handled by dsm_integer (the integer required would be too large). Or if the user objected to inputting phone numbers as integers, without hyphens or parentheses, dsm_v2_astring could be used instead.

As a second example consider a data type to be called "date_start" as part of a relation called "job_history" with data types "name", "position", "date_start" and "date_end". Dates can be managed by dsm_v2_astring since dsm_v2_astring effectively accepts any character string datum. But dsm_date manages them much more efficiently. And dsm_date verifies that the data element supplied is actually a date: it accepts "3 jan 74", "1/3/74" or "January 3, 1974" as all representing the same date, and rejects "Jaxuary 3, 1974" or "31feb72" as invalid dates. So dsm_date: 1) is more efficient, 2) verifies data as it is inserted, 3) sorts dates chronologically whereas dsm_v2_astring sorts data elements alphabetically (i. e., "January 1, 1973", . . . ,"April 1, 1973" instead of "April 1, 1973", . . . ,"January 1, 1973") and 4) is more convenient ("2feb73" is much easier to type than "February 2. 1972" but is output the same).

If after creating a data type and inserting a good deal of information into it a user wants to have it managed by a different dsm, the data type can be "restructured". The process is not extremely complicated, nor is it too expensive. So the choice of a dsm should not be regarded as irrevocable. (See the documentation of "restructure_data_type".)

By convention all data strategy modules are given names starting with "dsm_". So commands which expect a dsm to be specified, such as new_data_type, allow this to be omitted:

new_data_type x dsm_integer

and

new_data_type x integer

are equivalent. In general, older dsm's have names ending without "_" (dsm_table) while newer dsms have a terminal "_" (dsm_decimal_). At some point all dsms will be required to have names beginning with "dsm_" and ending with "_".

Data types are classified as "virtual" or "non-virtual" depending on what type of dsm they are managed by. Data strategy modules such as dsm_integer and dsm_date, which translate a character string into a reference number directly, are called "virtual". Dsms such as dsm_v2_astring and dsm_table, which store the associations between character string and reference number in a file, are called "non-virtual". The "virtual" data types and dsms have the desirable quality that they use less storage and are usually faster at mapping character strings to reference numbers and vice versa.

A dsm may reject a data element for a number of reasons. In general, virtual data strategy modules reject data elements if they are in the wrong format (dsm_integer will reject "10" because "l" is non numeric, and should be "1", while dsm_date rejects "feb31" as an illegal date). Non-virtual dsms (those that require storage to maintain the association between reference numbers and character strings) reject a datum if there is no room for it. (But there are different ways in which different dsms can run out of room: one may run out of room if the file is full, another may fail if there is room left in the file but all the space in a particular part of the file is used up. For more specific reasons the user should see documentation of each data strategy module.)

The "exec_com" file "createdb.ec" can be used to create an RDMS data base. (See its documentation.) It inserts the standard RDMS data strategy modules into the data base. (Inserting a dsm does not mean copying it. Rather it can be used from the data base.) The standard dsms are: dsm_integer, dsm_v2_astring (alphabetical strings, this is version 2 of the dsm), dsm_table (for data such as ranks where "Department Head" comes before "Professor", i. e., the sort order is completely independent of the character string), dsm_date, and dsm_ciph_integer (data element can be retrieved from the reference number only when a password is supplied). A user may write his own dsm for some special application. This can be inserted into the data base by using "insert_module".

A reader who is interested in a discussion of how data strategies are implemented in RDMS is referred to the "RDMS Design Principles", section 1.6, which describes how data strategy modules interact with other modules of the system.

(END)

An Overview of Data Format Modules
09/24/74

A data format module (dfm) is a program which converts from one representation of data to another. It takes as input the reference number and character string representations of a data element, and modifies the character string to have the desired format. Some dfms do not use the input reference number (such as dfm_names_, which merely reshuffles the parts of the character string into another order) while other dfms do not use the character string (dfm_abbrev_date_ uses the reference number of a date to produce the character string). All data format modules modify the character string, setting it to the desired value, but the data type file from which the datum was obtained is not modified (the character string is only a copy of data in the data type file).

Each data strategy module (dsm) has its own standard format for output character strings, even though some dsms accept various input formats. For example, dsm_date will accept "03/29/74", "29 mar" or "March 29, 1974" as all indicating the same date. But when asked to produce a character string suitable for printing, dsm_date always returns "March 29, 1974" to represent that particular date.

A data format module complements the data strategy module. Whereas the dsm accepts data in a format most convenient for input, the dfm produces data formated appropriately for output. Suppose that only the month and year in which various dates fall were needed for a specific report. Then dfm_mmyy_, which produces output strings such as "03/74" and "11/70" would be appropriate.

Usage:

Several RDMS commands and report generating programs allow the user to specify that a dfm be used for a particular data type or data types. The relational editor "eds" has a Format request which allows the user to specify that some dfm will be invoked when printing a data element. Report programs such as "quick_report", "book" or "sharri" have their own ways of specifying what dfm to use. Also, the "list_data_type" command allows the user to specify a dfm to be used when printing the contents of a data type.

Notes:

Some data format modules currently availible include:

dfm_abbrev_date_
: This dfm produces dates such as 9/13/73. (The data strategy module dsm_date produces dates such as "September 11, 1973".)

dfm_abbrev_name_
: This data format module assumes names are stored in a form such as "McGary, Thomas B." (i. e., last name followed by a comma and one space followed by first name and possibly a space and initial or initials.) Then the dfm will produce last name followed by initials. (e. g., "McGary, T. B.")

dfm_get_initials_
: Extracts initials of first, middle and last name; or if no middle name, then initials of first and last names.

dfm_last_name_
: Extracts last name only.

dfm_mmyy_
: Produces month and year of date in form 9/73 or 10/70.

dfm_names_
: Rearranges parts of a name so that first name occurs first, followed by middle initial and last name. (Names are normally stored with the last name first, followed by a comma and the first and middle names.)

dfm_null_string_
: Always produces a null string, which is one method of suppressing the printing of a data type.

dfm_parenthesize_
: Encloses data element in parentheses ("(" and ")").

dfm_percent_
: Appends "%".

dfm_phone_numbers_
: Prints dsm_integer data type as phone number, allows M.I.T. extension ("X-4107"), Centrex numbers ("3-4107", "8-7700"), local numbers ("253-1000") and numbers with area code ("(617) 253-1000").

dfm_right_justify_
: blank pads on the left to fill field of specified size. (field size chosen by entrypoint, dfm_right_justify10_ for field of 10 characters, etc.) If string is wider than field it is trunacted.

dfm_soc_sec_num_
: Prints dsm_integer data type as social security number, 123456789 -> 123-45-6789.

For specific details on these and other data format modules, see the writeups on individual dfms.

Naming Conventions:

Commands and subroutines that use dfms call the program "get_dfm_ptr_" to find the entrypoint desired. The program requires that all data format modules be given names which start with "dfm_" and end with "_". Thus any name given to get_dfm_ptr_ is first expanded to meet this requirement:

mmyy	becomes	dfm_mmyy_
dfm_mmyy	becomes	dfm_mmyy_
mmyy_	becomes	dfm_mmyy_

Next get_dfm_ptr_ attempts to find a segment with this name, using the "search rules" for object segments. (See The Multics Programmers' Manual Part II: REFERENCE GUIDE TO MULTICS, Section 3.2 "The System Libraries and Search Rules".)

If such a segment is found, and an entrypoint of the same name can also be found in that segment, the search succeds and stops here. Otherwise, the segment named "bound_dfms_" in the RDMS service directory is checked for the entrypoint. (This segment contains all the dfms mentioned in the Note above.) Otherwise the search fails.

If the name specifying the dfm contains a "$", get_dfm_ptr_ assumes the name contains a segment name and an entrypoint name ("my_dfms_$dfm_roman_numeral_" would specify the entrypoint "dfm_roman_numeral_" in the segment "my_dfms_".) Data format modules must always have names beginning with "dfm_" and ending with "_", so if the name supplied was "my_dfms_$roman_numeral", get_dfm_ptr_ would expand the entrypoint name as above and thus look for "my_dfms_$dfm_roman_numeral_".

If the specification contains any "<" or ">"s, get_dfm_ptr_ assumes the directory containing the dfm is specifed and only searches that directory. For example, given ">udd>Proj>Name >dfm_dir>mydfms_$dfm_roman_numeral_", get_dfm_ptr_ will search only the directory >udd>Proj>Name>dfm_dir for a segment named "my_dfms_".

Example:

The "list_data_type" (ldt) command allows the user to specify that a dfm be used in printing the contents of a data type. Suppose the name data type in the currently set data base contains only a few names. These may be listed by the command

```
ldt name
```

which would produce the following response:

```
                    name


          4294967296  Adler, Mida E.
          8589934592  Arnold, Theodore
         12884901888  Atkins, James D.
         17179869184  Austin, Linda
         21474836480  Babcock, Bruce
         25769803776  Barber III, John B.
```

In order to list the same data type using dfm_names_, a user might type:

```
ldt name -dfm names
```

The following would result:

```
                    name


          4294967296  Mida E. Adler
          8589934592  Theodore Arnold
         12884901888  James D. Atkins
         17179869184  Linda Austin
         21474836480  Bruce Babcock
         25769803776  John B. Barber III
```

(END)

add_name_set

Command
09/24/74

Names: add_name_set,
ans,
add_name_set_force,
ansf

Entry: add_name_set,
ans

The "add_name_set" command is used to add a name to a data-type or relation which already exists in the user's data-base. This permits a relation or data-type to be referenced by more than one name. (In a departmental data-base, for example, "name" and "svsr" refer to the same file of people's names.)

Usage:

The command may be invoked by typing:

add_name_set oldname newname

or

ans oldname newname

Either form of the command will add the identifier newname to the file (relation or data-type) identified by oldname.

Note:

If newname is already present on an existing relation or data-type, the command will print an error message and the name will not be added.

Entry: add_name_set_force,
ansf

This entry is used to add a name to an existing relation or data type, but if a set with the new name already exists in the data base, the name will be removed. (When the last name on a set is removed, the set is deleted.)

Usage:

add_name_set_force oldname newname

or

anf oldname newname

Note:

If there is already a set named newname in the data base, the new name will still be added; this forces the removal of newname from the already existing set, even if it is the last name on that set. Removal of the last name on a set automatically deletes the set. Thus the user is advised to be very cautious when using this entry.

(END)

change_name_set

Command
09/24/74

Names: change_name_set,
cns,
change_name_set_force,
cnsf

Entry: change_name_set
cns

This command renames a relation or data type with a new name.

Usage:

change_name_set *oldname newname*
or
cns *oldname newname*

Example:

```
change_name_set phone_book phone_book.old
```

Note:

If a relation or set with name *newname* already exists in the data base, an error message is printed and the name will not be changed.

Entry: change_name_set_force,
cnsf

This entry renames a set, but instead of printing an error message if a set named *newname* already exists, the set named *newname* will be deleted before changing the name *oldname* to *newname*.

Usage:

change_name_set_force oldname newname

or

cnsf oldname newname

Note:

This entry will delete newname if it already exists in the data base and then rename oldname to newname. Caution is advised when using this entrypoint.

(END)

check_info_segs

Command
09/24/74

Names:

check_info_segs
cis

Purpose:

This command is similar to the Multics command of the same name, and is used to determine if any info files have been modified since its last call. But, in addition to checking the Multics help directories, this version checks the RDMS help files also.

Entry:

check_info_segs
cis

Usage:

The command line:

check_info_segs

or

cis

checks the date and time modified of each info file in the Multics directories >documentation>info_segs and >documentation>lmi_info_segs, and also the RDMS info directory >udd>RDMS>info_segs. If the date and time modified is more recent than the date stored in the user's abbrev profile (the segment is >udd>Project>User>User.profile) the name of the info file is printed.

The command also accepts arguments specifying what directories will be searched and what action will be taken when a changed info seg is encountered.

The extened use is:

check_info_segs -ctl-args-

or

cis -ctl-args-

where -ctl-args- is optional, and may include any of the following:

-date date-time
-dt date-time

Instead of using the date and time stored in the user's profile, the date time supplied will be used. date-time can be any specification of date and time acceptible to convert_date_to_binary_ (such as "9 AM November 17, 1973" or "0900.0 11/17/73"; "1900 11/17/73" is not accepted because "1900" could be a year or a military time, so "1900.0" is used to specify military time). The date-time stored in the user's profile is not updated.

-call command-line

For each info file that has changed, the full pathname of the segment is appended to command-line, forming a command line which is submitted to the command processor. Most frequently the command line selected is "help -pn" (the full command line might be:

check_info_segs -call "help -pn"

or the command line could contain additional arguments) to invoke the help command for each changed file. (A blank is inserted between the command line supplied by the caller and the pathname of the info file. Thus if >udd>RDMS>info_segments>eds_changes.info had been modified since the last check, the command line above would result in the line:

help -pn >udd>RDMS>info_segments>eds_changes.info

being submitted to the command processor.)

-no_update
-nud

The date and time stored in the user's profile will not be updated. (Normally the date and time are updated so that when called at some later time check_info_segs will report only files that have changed since the last check.)

-long
-lg

The date-time modified and the name of the help file will be printed on the console. By default, in "normal" mode only the name of the changed help file is printed.

-brief
-bf

The name and date-time modified will not be printed on the console. This control argument is useful in conjunction with the "-call" argument.

-pathname directory>star-name
-pn directory>star-name

The Multics and RDMS help directories will not be checked for info files that have changed; instead all files in directory whose names match star-name will be checked. This argument may occur more than once, in which case each directory specified will be checked. (Example: to learn what info segs in directory have changed, use a star-name of "**.info", meaning all segments with names ending in ".info". To check all segments in directory use a star-name of "**".)

Example:

The command line:

```
check_info_segs
```

Might result in:

```
check_info_segs:  Checking >doc>info>**.info.
pl1_changes.info
pl1_status.info

check_info_segs:
  Checking >udd>RDMS>info_segs>**.info.
eds.info
eds_changes.info
```

This indicates that the Multics PL/1 compiler has been changed, and information about the changes can be obtained by typing "help pl1 pl1_status" and "help pl1_changes". Additionally, the RDMS relational editor, "eds", has been changed, and "help eds" and "help eds_changes" can be used to see what has been changed.

If the user knew he would be interested in all the info segs that had changed,

```
check_info_segs -call "help -pn"
```

would call help for each info seg.

Notes:

If the user quits while check_info_segs is listing the segments that have changed in some directory, the user may type "program_interrupt" or "pi" and check_info_segs will stop listing changes in that directory and move on to check the next directory. However, if the user "quits" during a call to some program (such as help) which was invoked by check_info_segs, a "program_interrupt" in this situation will cause check_info_segs to execute the command line for the next info seg in the directory it was checking. Thus, if the user does not wish to see all of a help file he may quit and move on to the next help file by typing "pi".

(END)

cleanup_data_base

Command
09/24/74

Name: cleanup_data_base
cudb

This command cleans up a data base directory by deleting quarts and temporary relations. It does not terminate the data base. (Many ordinary uses of a data base produce temporary relations and quarts.) On rare occasions, the number of temporary files may fill up all available slots for relations or quarts. Thus if there are too many quarts, the command can be used if new quarts need to be created and there is no room during the current process (since quarts are automatically deleted when a user logs out). Temporary relations must be explicitly deleted.

Usage:

```
cleanup_data_base
cudb
```

Example:

```
cleanup_data_base
cudb
```

Note:

If the user wishes to clean up and terminate a data base then he should use the command terminate_data_base.

(END)

compare_sets

Command
09/24/74

Names: compare_sets,
cos

This command compares the contents of two relations and prints out the differences between the two copies. It was designed to determine which copy of a relation is the most complete.

Usage:

```
compare_sets   relation-name-1     relation-name-2
```

Where relation-name-1 and relation-name-2 are the names or reference numbers of the relations to be compared.

Example:

```
compare_sets  phone_book eds.phone_book
```

Note:

Compare_sets uses the "difference" operation to determine the differences between relation-name-1 and relation-name-2. If a difference operation fails an error message is printed.

(END)

Command
09/24/74

Names:

copy_set
cps
copy_set_force
cpsf
insert_set
ins
insert_set_force
insf

These commands are used to add to a data base a set (data type or relation) which either resides in another data base or was created by other than the standard RDMS primitives.

Usage:

copy_set set-name sm-name
or
insert_set set-name sm-name

where:

1) set-name — is the name of the set. It must be less than 32 characters.

2) sm-name — is the name of the strategy module (data strategy module or relation strategy module) which should manage the segment associated with the data type. If this argument is omitted then the relation strategy module "rsm_matrix" is assumed. (The strategy module must be specified for data types.)

In the case of copy_set and copy_set_force, the set-name may or may not reside in the data base directory. If it does the command will fail unless copy_set_force has been used, in which case the segment previously existing in the data base directory will be replaced by the new segment. Copy_set (and copy_set_force) do not delete the segment specified by set-name, they merely copy it into a segment in the data base directory.

If insert_set (or insert_set_force) is invoked with a pathname, then a link is placed in the data base directory to the specified segment. The specified segment is not modified in any way. If a segment or link already exists in the data base directory with the entry_name portion of set-name then insert_set will fail. Insert_set_force will delete (or unlink) this already existing entry and replace it with the link to set-name.

Example:

```
ins >udd>RDMS>dis.shared>rank dsm_table
```

This adds to the data base directory a link named "rank" to a data type segment in the directory ">udd>RDMS>dis.shared". This data type will be managed by the data strategy module "dsm_table". The actual data associated with the data type rank still resides in the segment in >udd>RDMS>dis.shared named rank.

```
cps >udd>RDMS>dis.shared>rank dsm_table
```

This command also adds a data type named "rank" to the data base, but it creates a copy of the segment >udd>RDMS>dis.shared>rank in the data base directory. This segment is named rank. If a segment (or link) named rank already exists in the data base the copy_set command will fail, while the copy_set_force command will replace it with the new segment.

Notes:

The prefix "dsm_" or "rsm_" on the name of the strategy module is required. Otherwise copy_set (or insert_set) will not accept it as a strategy module name.

If set-name is an entry-name (that is, it contains no <"s or >"s), then it is assumed to be an entry in the user's data-base directory. Otherwise, the pathname is expanded relative to the users working directory.

(END)

create_relation

Command
09/24/74

Names: create_relation,
crr

This command creates an empty relation having the data types supplied by the user in response to its questions.

Usage:

create_relation relation_name
or
crr relation_name

Example:

(user input is underlined)

```
create_relation phone_book
create_relation: number of columns= 4

create_relation: data-type is name

create_relation: data-type is room

create_relation: data-type is extension

create_relation: dsm is   integer

create_relation: data-type is city

create_relation: dsm is   v2_astring
```

The above example would result in the creation of a relation named "phone_book" containg 4 columns: name, room, extension and city. In the process of creating this new relation, two new data types were also created, namely, extension and city. When a new data type needs to be created create_relation asks the user for the name of the data strategy module (dsm) which is to manage the new data type.

Note:

Relations may also be created using the relational editor "eds", or by several entries in sms_interface.

(END)

createdb.ec

Exec Com Segment
09/24/74

Name: createdb.ec

This exec com file contains commands which create a Multics directory to be used as an RDMS data base, set the acl and initial acl for segments, create RDMS system segments and insert the standard relational strategy modules (rsms) and data strategy modules (dsms) into the data base.

Usage:

ec createdb path

or

ec createdb path user1.project1 ... userN.projectN

path specifies the data base directory by relative or absolute pathname. If the directory does not exist it will be created.

The second form is used when creating a data base for another user or users. Up to 7 userI.projectI's may be specified. These are used in setting the access control list and initial access control list for segments on the data base directory.

Example:

ec createdb data_base

This will create a data base directory immediately below the users working directory. (If the working directory was >udd>Project>User, the directory created will be >udd>Project>User>data_base.) As the commands in the execute command file are executed, the following messages will be printed on the console. The user need not worry about these messages, as long as the final message indicating successful creation and initialization of the data base is printed.

createdb: creating data base data_base at 11/08/73 1242.3 est Thu.

sdb: data_base has been initiated for McGary RDMS at 11/08/73 1242.4 est Thu.

createdb: finished creating and initializing data_base at 11/08/73 1242.5 est Thu.

Notes:

The user creating the data base must have sma access on the directory if it already exists, or if the directory does not exist, he must be able to create it.

Createdb initiates the new data base, terminating any previously initiated data base.

Createdb inserts the following data strategy modules in the data base:

integer
v2_astring
date
table
ciph_integer

The relation strategy module "rsm_matrix" is also inserted.

If a user is not using the RDMS version of the Multics exec_com (ec) command the full path of the ec file must be specified (e. g. "ec >udd>RDMS>service.ec>createdb data_base" to use the full path as of this writing).

(END)

Active Function
09/24/74

Names: dbd

This active function returns the pathname of the current data base or, if no data base is initiated, a null string is returned.

Usage:

[dbd]

Example:

```
loa_ [dbd]
```
(This would print a pathname, such as >udd>RDMS>dbs>tdb, on the console.)

```
cwd [dbd]
```

```
st [dbd]>** -dt
```

(This might be used to determine the date and time dumped for each segment in the data base directory.)

(END)

delete_name_set

Command
09/24/74

Names: delete_name_set
dns
delete_name_set_force
dnsf

The delete_name_set command is used to remove a name from a set (relation or data-type) which has two or more names on it.

Entry: delete_name_set
dns

Usage:

The command may be invoked by typing:

delete_name_set name

or

dns name

Either form of the command will remove the identifier name from a relation or data-type in the user's current data-base. This command will not delete a set; it only removes a name by which the set is known. (The "delete_set" command may be used to remove a relation or data type from a data base.)

Note:

If name is the only name by which the relation or data-type is identified, the "delete_name_set" command will not remove name. (Refer to delete_name_set_force.)

Entry: delete_name_set_force
dnsf

This entry deletes the name on a set even if it is the last name on the set (in which case the set will also be deleted).

Usage:

delete_name_set_force name

or

dnsf name

Note:

If the name <u>name</u> is the only name on the set, it will still be deleted. Use this command with caution, since it may also delete the set from which the name is being removed, if <u>name</u> is the only name on the set.

(END)

delete_sets

Command
09/24/74

The delete_sets command causes specified relations and data_types to be deleted from the currently initialized data-base.

Names: delete_sets
dls

Usage:

```
delete_sets  set1 set2....setn
     or
dls set1 set2 . . . set3
```

set*i* is the name or reference number of a relation or data type in the currently initialized data base.

Notes:

The user must have modify access on the data-base directory.

The star convention may be used.

Example:

```
dls  eds.**  old.namelist  +TEMP+.*
```

deletes the set "old.namelist", all sets whose first name component is "eds", and all sets whose names have exactly two components, the first of which is "+TEMP+".

(END)

dfm_abbrev_date_

Data Format Module
09/24/74

Names:

dfm_abbrev_date_

This data format module produces a date in the form:

<month-number>/<day-number>/<year-number>

(e. g., "9/15/73"). This form of date is much more compact, and is always between 6 and 8 characters long.

Example:

dsm_date form	dfm_abbrev_date_ form
January 1, 1973	1/1/73
October 21, 1971	10/21/73
December 7, 1974	12/7/74

Notes:

This data format module uses the reference number to compute the string used to overwrite the input character string.

See also the overview on data format modules for a general discussion of data format modules and a list of those currently available.

(END)

dfm_abbrev_name_

Data Format Module
09/24/74

Names:

dfm_abbrev_name_

This data format module modifies a name in standard form (last name followed by comma, followed by first and middle names) to be the last name followed by commas followed by the initials of first and middle names.

Example:

An input bvs of "Albequerque, Alfredo Santiago" is modified in place to result in "Albequerque, A. S.".

"McGary, Thomas B." bcomes "McGary, T. B."

"Goldman, Jay" becomes "Goldman, J."

Notes:

The input string is modified in place and the reference number is not used.

See the overview on data format modules, which contains a list of currently available data format module and a general discussion of them.

(END)

Data Format Module
09/24/74

Names: dfms_$dfm_columns_
dfms_$dfm_terminate_columns_

Purpose:

This dfm formats data elements into fixed width strings, by padding on the right with blanks. The dfm can be used for formatting several different data types in a somewhat complicated manner. When first called the program asks the user for the number of fields and the width of each field. Then when next called it uses the width for the first field; the second time called it uses the width for the second field, and so on, until the nth time called the nth field width is used, and an internal counter is set to 1 so that the next cycle of calls will use the first through nth field widths. An entry is provided to reset the internal counter and force the following call to use the first field width, the entry being dfms_dfm_terminate_columns. @

Usage:

```
dcl       dfms_dfm_columns_   ext entry(ptr,fixed bin(35));
dcl       bvs_ptr             ptr;
dcl       bvs                 char(N) varying based (bvs_ptr);
dcl       refno               fixed bin(35);

call dfms_$dfm_columns_(bvs_ptr,refno);
```

At the time of the call, bvs_ptr has been set to point to some based varying string, perhaps as a result of a call to gd_. refno has been obtained from a relation or possibly by a call to gr_, but refno is not used.

If this is the first call, the number of fields and the width of each is obtained by querying the user.

Example:

(fill in later)

Notes:

This dfm is not recommended for the casual user.

(END)

dfm_commas_

Data Format Module
09/24/74

Name: dfm_commas_

This data format module inserts commas (",") into a numeric string. A comma is placed every three (3) characters starting from the right. If the character string is the null string then it is not modified.

Example:

"123" becomes "123" "1234" becomes "1,234" "" becomes ""

Notes:

If the string is not the null string then it is always replaced. The new format is computed directly from the reference number and correct results will occur only if the reference number was assigned by dsm_integer. Thus, this data format module can only be used to format datums from data-types managed by the data strategy module dsm_intger.

Refer to the overview on data format modules for a more detailed discussion of the use of dfms.

(END)

dfm_commas_decimal_

Data Format Module
09/24/74

Name: dfm_commas_decimal_

This data format module inserts commas (",") and a single decimal point (".") into a numeric character string. It is assumed that the datum being reformated represents a decimal number with two (2) digits to the right of the decimal point. If the datum is the null string then it is not modified.

Example:

"1234.00" becomes "1,234.00"
"" becomes ""
"123345.45" becomes "123,345.45"

Notes:

If the datum is not the null string then it is always replaced. The new format string is computed directly from the reference number. Correct results will occur only if the reference number was assigned by dsm_decimal. Thus, this data format module can only be used to format datums from a data-type managed by dsm_decimal_ and which allows two (2) decimal places to the right of the decimal point (refer to description of dsm_decimal_).

Refer to the overview on data format modules for a more detailed discussion of the use of dfms.

(END)

dfm_credit_card_

Data Format Module
09/24/74

Name: dfm_credit_card_

This data format module inserts dashes ("-") into a numeric character string after the third and seventh characters. It is used when the numeric string is a credit card number.

Example:

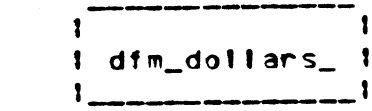
"1234567987" becomes "123-4567-987"
"" becomes ""
"123456789" becomes "012-3456-789"

Notes:

If the character string is not the null string it is always replaced. The new format string is computed directly from the reference number and correct results will occur only if the reference number was assigned by dsm_integer. Thus, this data-format module can only be used to format datums from data-types managed by dsm_integer.

Refer to the overview on data format modules for a more detailed discussion of the use of dfms.

(END)



Data Format Module
09/24/74

Name: dfm_dollars_

This data format module inserts a leading dollar sign ("$") and commas (",") into a numeric character string. Leading blanks are removed and commas are placed every three (3) characters starting from the right.

If the character string is the null string it is not modified.

Example:

"1234" becomes "$1,234" "" becomes "" "23" becomes "$23"

Notes:

The string is always modified in place. The new format is computed directly from the reference number. Thus, this data format module will only work correctly for reference numbers assigned by the strategy module "dsm_integer" (refer to dsm_integer elsewhere).

Refer to the overview on data format modules for a more detailed discussion of the use of dfms.

(END)

dfm_dollars_decimal_

Data Format Module
09/24/74

Name: dfm_dollars_decimal_

This data format module inserts a leading dollar sign ("$"), commas (",") and a single decimal point (".") into a numeric character string. It is assumed that the datum being reformated represents a decimal number with two (2) digits to the right of the decimal point. Leading blanks are removed and commas are placed every three (3) characters starting after the decimal point (which is placed to the left of the rightmost 2 digits). If the datum is the null string then it is not modified.

Example:

"1234.00" becomes "$1,234.00"
"" becomes ""
"123345.45" becomes "$123,345.45"

Note:

The string is always modified in place. The new format is computed directly from the reference number. Thus, this data format module will work correctly for reference numbers assigned by the strategy module "dsm_decimal_". This data format module can only be used to format datums from a data-type managed by dsm_decimal_ (and allowing two (2) decimal places to the right of the decimal point; refer to the description of dsm_decimal_).

(END)

dfm_get_initials_

Data Format Module
09/24/74

Names:

dfm_get_initials_

A standard representation of a name (last name followed by comma, followed by first name and middle names) is converted into two or more initials, the initial of the first name being first, the initials of middle names being next and the initial of the last name occuring last.

Example:

Standard Name	dfm_get_initials_ form
Albequerque, Alfredo Santiago	ASA
Goldman, Jay	JG
McGary, Thomas B.	TBM

Notes:

The character string containing the name is modified in place and the reference number is not used.

See also the overview on data format modules for a discussion of data format modules in general, as well as a list of currently available dfms.

(END)

dfm_last_name_

Data Format Module
09/24/74

Names: dfm_last_name_

This dfm extracts the last name from a complete name in standard form (i. e., "<last name>, <first name> <initial>.")

Example:

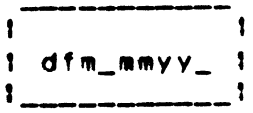
"Albequerque, Alfredo S." becomes "Albequerque"
"Goldman, Jay" becomes "Goldman"

Notes:

This dfm gets the parts of a name from its input string, rearranges them and then overwrites the input string. It does not use the reference number at all.

See the overview on data format modules for more detailed discussion of data format modules.

(END)



Data Format Module
09/24/74

Name:

dfm_mmyy_

This dfm abbreviates a standard date (i.e., "<month name> <day number>, <year number>") in the form "<mm>/<yy>", where <mm> is the month number (1 to 12) and <yy> is a two digit year number (01 to 99).

Example:

"September 5, 1971"	becomes	"9/71"
"January 15, 1973"	becomes	"1/73"

Notes:

This dfm uses the input reference number to compute the string it outputs. The input string is overwritten with the computed string, so its value need not be set before dfm_mmyy_ is invoked.

See also the overview on data format modules for a list of currently available data format modules and more detail on the use of dfms.

(END)

dfm_names_

Data Format Module
09/24/74

Names:

dfm_names_

A standard name in the form:

"<last name>, <first name> <middle initial>"

is transformed into :

"<first name> <middle initial> <last name>".

Example:

"Albequerque, Alfredo S."
becomes
"Alfredo S. Albequerque"

"Goldman, Jay"
becomes
"Jay Goldman"

Notes:

This dfm operates only on the character string datum -- it does not use the reference number.

See also the overview on data format modules for a more detailed discussion of dfms.

(END)

dfm_null_string_

Data Format Module
09/24/74

Names:

dfm_null_string_

An input character string is modified in place to be a null string.

Example:

"Albequerque, Alfredo S." becomes ""
"Goldman, Jay" becomes ""

Notes:

Only the character string is referenced. (This dfm does not use the reference number supplied when it is invoked.)

See also the overview on data format modules for a more detailed discussion of the use of dfms.

(END)

dfm_parenthesize_

Data Format Module
09/24/74

Names:

dfm_parenthesize_

Purpose:

An input string is enclosed in parentheses.

Example:

"no information" becomes "(no information)"

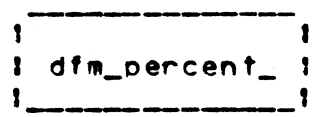
"" (a null string) is not changed.

Notes:

This dfm operates only on the character string datum (i. e., the reference number is not used).

Refer to the overview on data format modules for a detailed discussion of the use of dfms.

(END)



Data Format Module
09/24/74

Names: dfm_percent_

This data format module appends a percent sign after the string provided as input.

Example:

"100" becomes "100%"
"55" becomes "55%"
"" (a null string) remains ""

Notes:

This dfm operates only on the character string datum, not its reference number.

Refer to the overview on data format modules for a more detailed discussion of the use of data format modules.

(END)

```
+--------------------+
|                    |
| dfm_phone_numbers_ |
|____________________|
```

Data Format Module
09/24/74

Names:

dfm_phone_numbers_

This data format module expects a character string representing an integer as input, and reformats this string to represent a phone number.

This dfm operates only on the character string datum, and does not use its reference number.

If the character string is a null string it is not modified.

If the character string is less than five characters in length it is interpreted as an MIT extension, so it is prefaced with "X-".

If the character string is exactly five characters in length it is interpreted as a Centrex number. A dash is added between the first digit and the last four.

If the character string is longer than five characters but less than 8 in length, it is interpreted as a regular seven digit phone number and a dash is inserted after the first three characters.

If there are more than eight digits, the number is interpreted as a phone number with area code. The first three digits are enclosed in parentheses, followed by a blank. A dash is inserted before the last four characters.

Example:

A null string ("") is not modified.

"1234" becomes "X-1234"
"4107" becomes "X-4107"

"34107" becomes "3-4107"
"87700" becomes "8-7700"

"2534107" becomes "253-4107"
"2587700" becomes "258-7700"

"6172534107" becomes "(617) 253-4107"

"6172587700" becomes "(617) 258-7700"

Notes:

This dfm is intended for use with a data type managed by dsm_integer. That data strategy module converts character strings representing integers into reference numbers whose value is that integer, and when the character string is desired, dsm_integer converts the reference number into a character string representing that integer.

A more detailed description of the use of dfms can be found in the overview on data format modules.

(END)

dfm_right_justify_

Data Format Module
09/24/74

Names:

dfm_right_justify_
dfm_right_justify2_
dfm_right_justify3_
dfm_right_justify4_
dfm_right_justify5_
dfm_right_justify6_
dfm_right_justify7_
dfm_right_justify8_
dfm_right_justify9_
dfm_right_justify10_
dfm_right_justify11_
dfm_right_justify12_
dfm_right_justify13_
dfm_right_justify14_
dfm_right_justify15_
dfm_right_justify16_
dfm_right_justify17_
dfm_right_justify18_
dfm_right_justify19_
dfm_right_justify20_

The input string is padded with blanks on the left to make it as large as the field desired. The field width is specified by selecting the appropriate entry. (For a field width of 10 characters, dfm_right_justify10_ is used, etc.)

If the length of the character string is less than that of the field specified in the name of the entrypoint, the character string is padded with blanks on the left.

If the length of the input string is greater than the field width, the input string is truncated to the desired length.

Example:

For a field width of 10 (the call will be made to dfm_right_justify10_) an input string in the first column produces an output string in the second column:

"1234567890"	"1234567890"	(not modified)
"12345"	" 12345"	(five blanks added)
"12345678901"	"1234567890"	(truncated to 10 chars)

Notes:

This dfm modifies its input character string in place without using the reference number.

More information on dfms may be found in the overview on data format modules.

(END)

Data Format Module
09/24/74

Names:

dfm_soc_sec_num_

Purpose:

This data fomat module transforms character strings representing integers into the form used for social security numbers.

The string will be modified in place. The reference number is not used.

If the string is the null string it is not modified.

If the string is less than seven characters it is interpreted as an M.I.T. registrar's number (for foreign students usually). A capital "R" is inserted before the string.

If the string is more than seven characters in length it is interpreted as a normal social security number. Dashes are inserted after the third and fifth digits.

Example:

"123456" becomes "R123456"

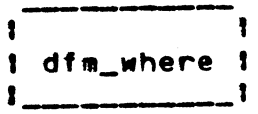
"123456789" becomes "123-45-6789"

Notes:

The string is always modified in place without using the reference number.

Refer to the overview on data format modules for a more detailed discussion of the use of dfms.

(END)



Command/Activ-Function
09/24/74

Name: dfm_where
dwh

This command searches for data format module entrypoints using get_dfm_ptr_, and returns the pathname of the segment in which the entry is found or reports that the search failed.

Usage:

dfm_where name1 . . . nameN
or
dwh name1 . . . nameN

Each namei identifies a dfm entrypoint. For "dfm_x_", namei may be "x", "dfm_x", "dfm_x_", or "x_", since by convention all dfm's are named in a similar way.

The command prints the pathnames, while the active funtion returns a character string formed by concatenating all the paths, separated by blanks.

Example:

dfm_where mmyy

Result:

>udd>RDMS>service>bound_dfms_$dfm_mmyy

will be typed on the console.

Note:

Refer to the overview on data format modules for more details on the use of data format modules.

(END)

display_relation

Command
09/24/74

Name: display_relation, dr

display_relation is used to print the contents of a relation. It allows much more control over formating than either quick_report or print_set does, but does not generate headers and footers like quick_report. It also allows the suppression of duplicate entries in the leftmost columns of a relation.

Usage:

display_relation relation-name -options-

where:

relation-name is the name of the relation to be printed.

-options- is one or more of the following options.

The options are divided into two (2) categories, namely control arguments which affect the printing of certain information concerning the relation, and those arguments which affect the format in which the actual information in the relation is displayed.

The control arguments are:

-brief
-bf
this arguement suppresses the following information: the name of the relation, the name of the strategy module, the reference number of the relation, the cardinality and arity of the relation.

-no_info
-ni
this argument suppresses all of the information listed for the -brief control argument except the sort order.

-no_sort
-ns
this argument suppresses the sort order of the relation.

-rows R
-r R — R specifies the number of rows of the relation to be printed. If this argument is omitted then all rows after the starting row up to and including the ending row will be printed.

-from
-from row — the relation will be printed starting at the row°th row. If this argument is omitted then printing will start at the first row of the relation.

-to row — the relation will be printed up to and including the row°th row.

The following arguments affect the printing of all the information in the relation:

-character
-ch — the information in the relation will be printed in datum form. This is the default mode if no output mode is specified.

-decimal
-dec — the output mode will be set to decimal. The information in the relation will be printed with the reference numbers for each datum expressed in decimal.

-octal
-oct — the output mode will be set to octal. The information in the relation will be printed with the reference numbers for each datum expressed in octal.

-no_duplications
-nd — this argument specifies that the leftmost column(s) of one row which match the immediately previous row are not printed. Instead, a null string becomes the value

printed for the column of the relation. Thus in a relation sorted on name, if a person has more that one row in the relation the person's name would be printed only once. Refer to the examples for more uses of this control argument.

-break BRK
-brk BRK
: this argument sets the default ioa_ control string inserted between columns of the relation. The default ioa_ control string is inserted between two columns of the relation when neither a column width for the first column nor a literal string occurs between the columns. The default ioa_ control string is initially " ". This value (BRK) for the ioa_ control string can be overridden at a later point in the command line by another occurrance of the -break control argument.

The following arguments specify the order of printing of the columns of the relation as well as controlling the format of each datum.

-data-type DI
-dt DI
: this arguments indicates that the data-type DI should be printed. When this argument is found, the ioa_ control string formating the output of each row of the relation is updated to contain the break string between this data-type and the previous one.

  If a literal string argument (refer to ARG below) occured between the previous occurance of the -data-type argument and this one, the default ioa_ control string (the break string) is not added; otherwise, (i. e., when no separation between the columns has been specified) the default ioa_ control string is appended to the output ioa_ control string.

-col COL
: this argument is similar to the -data-type argument except that the column to be output is specified by column number instead of data-type name. The updating of the output ioa_ control

string is the same as above.

-dfm DFM — this argument specifies that the data format module DFM should be applied to datums from the data-type specified by the most recent occurrance of a -data-type or -col argument.

-width W
-w W — this argument specifies that datums for the data-type specified by the most recent occurrance of a -data-type or -col argument should be left justified in a field of width W charactrers. The occurrance of this argument inhibits (in the same manner as a literal string (ARG below)) the use of the default ioa_ string at the next occurrance of the -data_type or -col argument.

-use_data-type DT
-use_dt DT — this argument specifies that the reference numbers for datums in the data-type specified by the most recent occurrance of a -data-type or -col argument should be looked up in the data-type DT instead of the one specified in the relation.

If the user wishes to have more control over the format of the relation than the -width and -brk arguments provide; the following arguments may be used:

-ctl IOA — IOA becomes the output ioa_ control string. The -width, -break and any literal string arguments are ignored. The output ioa control string must contain a "~a" for each data-type (column) of the relation to be printed.

ARG — ARG is any non-control argument or argument not otherwise used for any control argument. It is appended to the output ioa_ control string as a literal character string and inhibits the use of the default ioa_ control string (the break string) when the next -data-type or -col argument is reached.

Examples:

```
display_relation phone_book ~/ -dt name -w 30 -dt room
```

The above command would print the relation phone_book in the following format:

phone_book

matrix 284
cardinality 2, arity 3

name	room
Canning, H.	2-456
Gilbert, R.	10-250

If the name of the relation and size information was not desired the the -no_info (-ni) argument would be specified. If the sort order information was also not desired then the -brief conrol argument would be specified.

Note that the default ioa_ control string (the break string) was not used in this printout since both occurrances of the -dt control argument were preceded by either a literal string argument ("~/") or a -width (-w 30) argument.

The command:

```
display_relation phone_book -dt name -dt room
```

would print:

name	room
Canning, H.	2-456
Gilbert, R.	10-250

(END)

Data Strategy Module
09/24/74

Names: dsm_char4_

This data strategy module allows virtual storage of character strings of four or fewer characters. (A virtual data type is one in which the character strings are not stored in the data type, but rather the character string is produced as a function of the reference number. Virtual data types have the virtue of using very little storage: only enough space for a header, which all dsms are required to have.)

Usage:

When a data type is created it is necessary to specify its data strategy module. (See the discussion of the command "new_data_type.")

Example:

Using new_data_type, one proceeds as follows: (The lines typed by the user are preceeded by "->")

```
-> new_data_type xxx dsm_char4
   dsm_char4: Maximum number of characters allowed on
          input (1 to 4):
-> 3
   dsm_char4: Maximum number of characters allowed on
          output, allowing for dfm, is:
-> 12
   New Data Type xxx with refno 98 and strategy module
          dsm_char4_.
```

Notes:

This dsm knows the ascii character code and expects four 9 bit characters in a word. (In other words, it is Multics dependent.)

An attempt to insert a character string datum into a data type managed by dsm_char4_ will fail only if the string is longer than the maximum length specified when the data type was created.

(END)

Data Strategy Module
09/24/74

Name: dsm_char5_

dsm_char5_ is similar in purpose to dsm_char4_. It is used to store reference numbers to character string datums of maximum length five (5). Since it is possible to store five ASCII characters in a single word (the size of a reference number), dsm_char5_ is a virtual data-type. However, in order to put five characters in a single word, dsm_char5_ limits itself to the present (128 character) ASCII character set. Since five character datums may be quite useful (such as zip codes) it is felt that this restriction is worth the efficiency of a virtual data type.

Usage:

To create a data type managed by dsm_char5_ the new_data_type command is used. dsm_char5_ does not request user input, since the maximum length of a datum is always five (5) characers.

Example:

New Data Type zipcode with refno 84 managed by dsm_char5_.

Notes:

The only time dsm_char5_ will reject a character string is when the length of the character string exceeds five (5).

Data format modules which operate on datums returned by dsm_char5_ are assured of a 32 character varying string in which to put the reformated datum.

(END)

dsm_date

Data Strategy Module
09/24/74

Name: dsm_date

dsm_date is used to assign reference numbers to datums which represent dates. The reference numbers preserve the chronological ordering of dates. Thus the reference number for "January 23, 1972" is less than the reference numbers assigned to later dates such as "Feb 3, 1972", "1-23-73" or "Jan 1, 1974". dsm_date allows for a wide variety of input formats (such as the examples of dates above) while always producing a standard output format.

Usage:

To create a data-type managed by dsm_date the new_data_type command is used. For example:

User: new_data_type start_date date

System: new_data_type: data type "start_date" created with reference number 139 and strategy module dsm_date.

The maximum length output string is 18 characters (corresponding to a date such as September 10, 1973). All dates are represented on output by the standard form "*month* *day*, *year*" where *month* is the name of the month with the first character capitalized, *day* is a one or two digit day of the month and *year* is a four digit year.

Example:

Once a data-type managed by dsm_date is created, it may be used in relations. When inserting a datum into the data-type a wide variety of input formats is accepted. The following are legal input formats:

1) January 23, 1973

2) January 23, 1973

3) Jan 23, 1973

4) 1 23 1974

5) 1 23 74

6) 23 jan 74

7) jan23,74

8) 1.23.74

9) 1.23
(When the year is omitted from a date, the year on which the date next falls is used. Thus if today's date is January 23, 1973, the year implied is 1974. Similarly, if today's date is later than January 23, 1973 but earlier than January 23, 1974, the year implied is 1974. But if the date were January 23, 1974, the year 1975 would be implied.)

All of the above strings will be output as "January 23, 1974", and are all assigned the same reference number by dsm_date.

This dsm can reject character strings for two reasons:

1) The character string was not in the format of a date ("1 23 19 74" or "january 23m 1974").

2) The character string was in the format of a date but represented a non-existant date ("February 30" or "January 45").

(END)

dsm_decimal_

Data Strategy Module
09/24/74

Name: dsm_decimal_

dsm_decimal_ is used to assign reference numbers to datums which are decimal values. Each data-type managed by dsm_decimal_ allows a fixed number of places to the right of the decimal point. For numbers between 1 and -1 a leading zero is usually supplied (e. g., "0.15" or "-0.27") but if the user desires the leading zero may be suppressed (e. g., ".15" or "-.27"). The reference number assigned to a datum is the numeric value of the datum ignoring the decimal point with zero (0) characters appended to force the number of characters to the right of the decimal point to be the fixed number for the data-type. Thus a data-type allowing two (2) decimal places would assign the reference number 1245 to the character string "12.45". The character strings "12", "12.", "12.0" and "12.00" would all be assigned the same reference number, namely 1200. The reference number 1200 would always be output as "12.00".

Usage:

To create a data-type managed by dsm_decimal_ the new_data_type (new_data_type) command is used. For example,

```
new_data_type percent decimal_
```

The "decimal_" specifies that dsm_decimal_ will be the strategy module. (The user might also have typed "dsm_decimal_".) The program new_data_type will invoke dsm_decimal_ to create a new data type named percent, and dsm_decimal_ will ask the questions below. Responses typed by the user are underlined.

```
dsm_decimal_: How many decimal places do you want (up
    to 8)? 3

dsm_decimal_: For numbers between -1 and 1 do you want
    the first zero to be printed? yes

new_data_type: data-type percent created managed by
    dsm_decimal_.
```

The percent data-type would allow 3 decimal places to the right of the decimal point and a leading zero (0) will be printed if the datum is between -1 and 1. Thus, the character string "-.3" would be assigned the reference number -30 and would be represented on output as "-0.30". If the first zero was not to be printed then the reference number -30 would be represented by "-.30".

Example:

```
new_data_type grade decimal_

dsm_decimal_: How many decimal places do you want (up
     to 8)? 4

dsm_decimal_: For numbers between -1 and 1 do you want
     the first zero to be printed? no

 New Data Type grade with refno 134 and strategy module
     dsm_decimal_.
```

Notes:

The maximum length of a character string datum is 32 characters.

In order to do meaningful arithmetic with the reference numbers assigned by dsm_decimal_, one must know the number of decimal places assigned to the data-type. Thus to multiply two dsm_decimal_ reference numbers one must divide the product by 10 raised to the nth power, where n is the number of decimal places. For example, to multiply two reference numbers for datums in the "percent" data-type, such as 1020 and 1510, one would divide the product by 100, obtaining the reference number 15402. In terms of datums this corresponds to multiplying "10.2" and "15.1" to obtain "154.02". The relational operators package provides built-in functions for manipulating reference numbers generated by dsm_decimal_.

This data strategy module will reject character string data elements if they are not in the form of decimal numbers. Thus a string containing a character besides a numeral or a decimal point will be rejected. In addition, a string with more than one decimal point or more than the maximum number of digits to the right of the decimal point is also illegal.

(END)

dsm_integer

Data Strategy Module
09/24/74

Name: dsm_integer

Usage:

Dsm_integer is a virtual data strategy module. It can be used to store datums which are entirely numeric. In many cases, dsm_integer can be used to store datums which are normally not printed as entirely numeric. For example, social security numbers can be stored using dsm_integer but can be reformatted using a data format module (dfm_soc_sec_num in this case) when printing a report. Thus if the datums of a data-type can be sorted by just their numeric characters and the non-numeric characters can be easily inserted for output, dsm_integer can be used.

Example:

Data-types which can usually be stored using dsm_integer include:

social security numbers
years
numbers of (days, hours, . . .)
percents (if integral)

Notes:

Since dsm_integer is a virtual data strategy module, it is cheaper than a non-virtual dsm.

This dsm rejects a character string if it does not represent an integer.

(END)

Name: dsm_table

dsm_table is used to assign reference numbers to datums when the ordering of the datums can not be computed algorithmically (e.g., alphabetically or numerically). Examples of such lists of datums are the months of the year, a list of salary types and the names of the academic terms.

Usage:

To create a data-type managed by dsm_table use the new_data_type command. For example:

```
new_data_type term table

dsm_table: The maximum number of datums for this data
     type is? 3

dsm_table: The maximum length of a datum is? 24

New Data Type term with refno 84 and strategy module
     dsm_table.
```

The data-type term has been created. It may contain up to 3 datums (which have not be specified yet) and each datum can be no more than 24 characters long. When specifing the maximum length of each datum one should also consider any data-format-modules which may be used to reformat the datums. The maximum length of a reformated datum must also be less than or equal to 24 characters.

To insert the datums into the table data-type one can use the relation editor, eds. The sort order of the datums of a table data type is the order of insertion.

Notes:

This dsm may reject a character string for several reasons:

1) If there is no room left in the table, i. e., the maximum number of datums specified when the data type was created have been inserted.

2) If the character string is longer than the maximum

length specified when the data type was created.

3) If the user does not have write ("w") access on the segment containing the binding between reference numbers and character strings, the new string can not be inserted.

(END)

dsm_v2_astring

Data Strategy Module
09/24/74

Name: dsm_v2_astring

dsm_v2_astring is a data strategy module for storing alphbetic datums. It is used when the sorting order for the datums is to be the ASCII collating sequence. The ASCII collating sequence sorts numerals before upper case letters and upper case letters before lower case letters. Special characters (such as a space, colon, semi-colon, etc.) precede the numerals. Thus, "Brown" is "less than" "brown" and "Gold, D." is "less than" "Goldstein, R." since a comma (",") precedes an "s" in the collating sequence. dsm_v2_astring assigns reference numbers to datums in a manner which preserves the order of datums. Thus, the reference number for "Gold, D." will be numerically less than the reference number for "Goldstein, R.".

Usage:

To create a data-type managed by dsm_v2_astring the new_data_type command is used. When creating a new data-type, dsm_v2_astring asks the user for the maximum length of a datum. This number does not affect the returned length of a datum; it only restricts the datums that can be inserted into the data-type. For example, if the maximum length is set to 10, the character string "Gold, R." would always be returned as a character string of length 8. The character string "Goldstein, R." could not be inserted into the data-type since its length exceeds the maximum length.

Example:

```
new_data_type name v2_astring

dsm_v2_astring: What is the maximum string length? 50

New Data Type name with refno 83 managed by
     dsm_v2_astring.
```

The above command creates a data-type named "name" with a maximum datum length of 50.

Notes:

The algorithm used to assign reference numbers to datums is highly sensitive to the order in which datums are inserted into the data-type. In fact, more than 35 datums can not be inserted in alphabetic order. Users should make sure that datums are never inserted in alphabetic sequence for more than two or three datums.

Data-types managed by dsm_v2_astring can become very inefficient for a number of reasons. If a number of datums are inserted in order, the searching algorithm for finding the datum associated with a given reference number becomes more expensive. If a significant number of the datums in the data-type are no longer being used in any relation (the datum was incorrect and the relation was edited to insert the correct value or the datum was no longer needed) the data-type becomes wasteful in terms of storage costs. In order to correct these problems the "restructure_data_type" command is provided. This command creates a new version of the data-type which only contains those datums which are present in a relation in the data base. This restructuring assigns new reference numbers to the datums, so restructure_data_type must update every relation which uses the data-type being restructured. Thus, restructuring tends to be expensive and should not be done too often.

A restructure of the data-type can also be done to correct an incorrect maximum length specification. When restructuring a data-type managed by dsm_v2_astring, the restructure_data_type command must be informed that the data type is an "astring data-type". This is specified by the "-astring" option to restructure_data_type. Refer to restructure_data_type elsewhere in this manual.

An attempt to insert a new datum into a data type managed by dsm_v2_astring may fail for the following reasons:

1) If the datum is longer than the maximum length specified when the data type was created, it is rejected outright.

2) If the data type has no reference number available for the new datum, the attempt will fail. (This can happen when several character strings alphabetically close to each other appear in the data type. For example, if "Alice" has refernce number 14 and "Arthur" has reference number 15,

"Allen" cannot be inserted because no reference number between 14 and 15 is available. This problem can be corrected by "restructuring" the data type. See the discussion of the restructure_data_type command.)

3) Finally, if a reference number is available dsm_v2_astring will attempt to add the association between the new datum and its reference number to the data type segment. This requires modifying the data type segment, so the user must have write ("w") access on the segment.

(END)

Name: eds

The eds command invokes the standard relation editor. It is used for creating and editing relations.

Usage:

eds relation file-name

where

relation is the name or reference number of the relation to be edited.

file-name is a relative or absolute pathname of an eds input file. If present, eds will start reading from this file (see the "g" request below) after any necessary relations have been created. Eds will be in either input or edit mode depending upon the existence of relation. Refer to the Notes for further explanation. If file-name is omitted, eds will be in non-g-mode.

Notes:

eds operates in response to "requests" from the user. These requests are discussed below. eds has three modes of operation -- edit mode, input mode, and increment mode. Requests are accepted in all three modes. In edit mode the request must be the first character of the line; in input mode the request must be prefaced by a "." (period), or it will be treated as an input line. Increment mode is a generalized extension of input mode. It allows any request to be repeatedly executed with arbitrary arguments. edit mode may be entered in two ways: if the relation already exists, edit mode is entered automatically when the eds command is invoked: if eds is in input mode, the mode-change request must be issued. The mode-change request is a "." (period) standing alone in the leftmost position on a line. The request announces its current mode by responding with either "EDIT", "INCREMENT" or "INPUT" when the mode is first entered. From edit mode, input mode may be entered by typing the mode change character followed by a carriage return. Increment mode is entered by using the + (increment) request (see below). Leaving increment mode is accomplished by typing the mode-change request followed by a carriage return.

Requests to eds treat a relation as a collection of lines in a table to which there is appended a column of numbers indicating the "number" of each line. The index is the number which specifies the current line; that is, the line available for modification or printing. Some requests cause the index to be changed; other requests operate upon the line indicated by the index in the table. Still others modify the relation as a whole or affect the internal state of eds. All requests are performed upon a working copy of the relation specified. Because of this, the relation identified by the invocation of eds is not modified until the user writes it. (See "w" request.)

Upon entering eds via the command

eds relation

the user's data-base is searched for a relation with the name relation. If found, eds then searches for a relation named eds.relation. If eds.relation exists, it is used as the working copy. Otherwise, eds.relation, a copy of relation, is created in the data-base. The user is informed whether eds is creating or using an already existing copy of relation. Also, eds informs the user of and differences between the cardinality (number of rows) or arity (number of columns) of relation and eds.relation. Eds then enters EDIT mode.

If relation does not exist then eds will ask the following questions. First, eds will ask for the number of columns; then, for each column, eds will ask for a data-type name. If the data-type specified does not exist in the user's data base then eds will ask for the data-strategy-module which should manage this data-type and create the data-type for the user. eds then creates eds.relation as described above and enters INPUT mode. (See the "." request.)

Example:

eds phone_book

The above command may result in a number of messages printed on the users console. If phone_book already exists in the data-base, but eds.phone_book does not, then eds will print:

eds: eds.phone_book will be created.

If eds.phone book exists then eds will print:

eds: eds.phone_book exists. It will be used.

If phone_book and eds.phone_book have different cardinality then eds prints:

eds: phone_book and eds.phone_book disagree in length,
eds.phone_book will be used.

If phone_book and eds.phone_book have different arity then eds prints:

eds: phone_book and eds.phone_book disagree in order,
eds.phone_book will be used.

eds will then enter EDIT mode and print EDIT on the users console.

If phone_book does not exist in the data base then the following dialogue might result: (user input is underlined)

RELATION NOT FOUND

eds: number of columns=3

eds: data-type is name

eds: data-type is room

eds: dsm is dsm_room

eds: data-type is extn

eds: dsm is dsm_intger

eds: Character string is not an element of data type.
dsm_intger is not a data-strategy module.

eds: dsm is integer

phone_book is now created, containing three columns, name, room and extn. In the process, two new data types were created, room with data strategy module dsm_room and extn with data strategy module dsm_integer. This example assumes that the data-type name already existed in the data base and thus eds did not need to create it. If eds does not have to create a data-type it does not need (and does not ask for) the data-type's data-strategy module. In some cases, data strategy modules will ask questions concerning a new data type. Refer to the commands

"new_data_type", "convert_data_type", and "restructure_data_type" for more information concerning the creation of data-types. Note that eds accepts data strategy module names with or without the prefix "dsm_". Since it is required by RDMS, eds will add the prefix if it is not provided.

Once phone_book has been created, eds will then check for eds.phone_book and will print the same messages as described above. eds will then enter INPUT mode and print INPUT on the users console. (See the "." request.)

REQUESTS TO EDS

Each request is indicated by a single character, which must be the first character of a "request"; it is followed by (in some instances, optional) arguments. The requests are:

"a" request (absolute line number)

Format:

an or ann

Purpose:

Change the value of the line index to n (that is, the n-th line counting from the top line which is always line number 1). If the request is terminated with "p", the new current line is printed.

Spacing:

Any number of spaces may appear between the request and n. An arbitrary number of spaces may also appear between n and p.

Default:

If n is absent, 1 is assumed.

Note:

If the request is given with an "=" in place of the n, the current line is set to the previously stored value of the current line. (See the "=" request) Note that the contents of the restored line may not be the same as it was when the line index was saved due to deletions, resorting or relational operations.

"b" request (bottom)

Format:

b or bp

Purpose:

Reset the line index to the last line of the current working copy. If the "p" is present, the last line of the current working copy (i.e., the new current line) is printed.

"c" request (change)

Format:

A) cn/data-type/string-1/string-2/

or

B) cn/string-1/string-2/

or

C) cn/data-type/string-1/

Purpose:

A) Replaces every occurrance of the substring string-1 in the column indicated by data-type with string-2. n is an optional integer which indicates the number of lines which the request will operate upon.

B) Replaces every occurance of the substring string-1 in the next n lines with string-2 irrespective of data-type (column).

C) Replaces in the next n lines the entry for the column indicated by data-type with string-2 irrespective of its current value.

Delimiters:
Any character not appearing in string-1, string-2 or data-type may be used as the delimiter. (A "/" is shown in the format examples above.)

Default:
If n is absent, 1 is assumed.

Index:
Unchanged even if n is present.

Note:
Format C is assumed if there are two fields and the first is a data type name appearing in the working copy. Thus format B cannot be used if string-1 is a data type name.

"C" request (compose)

Format:
C relation-name

Purpose:
Compose the current working copy with the given relation-name. relation-name may be either the name or reference number of a quart or a relation.

Spacing:
A space may appear between the request and the relation-name.

Note:
See the description of compose elsewhere in this reference guide for a complete explanation of the compose operation.

Index:
The line index is set to the first line of the resulting relation.

"d" request (delete)

Format:
dn or dnp

Purpose:
Delete the next n lines from the working copy of the relation. Deletion begins with the current line. If the request is terminated with the character "p", then

the new current line is printed upon the completion of the deletion request. If "p" is not present, the new current line is not printed.

Spacing:

Any number of spaces between the request and n may be present. An arbitrary number of spaces may also appear between n and "p".

Default:

If n is omitted, the current line is deleted.

Index:

The index is set to the next line after the last line deleted.

Note:

deleting a line has the effect of moving all the lines below the deleted line up one line. Thus, the request "dp" will delete the current line, and print the new current line.

"D" request (duplicate and delete)

Format:

A) Dn/data-type-1/string-1/.../data-type-m/string-m/

or

B) Dn/data-type/string-1/string-2/..../string-m/

Purpose:

Delete the current line after duplication. This request is identical to the " (ditto) request, except that the current lines are "replaced" at the current line numbers by the "dittoed" lines. The new lines are not written at the bottom of the relation. Refer to the ditto (") request for an example.

Index:

The current line index is unchanged.

Note:

The formats above can be intermixed in the same request. The first field must be the name of a data type present in the working copy; if it is not the D request fails. After the first field, each field is checked if it is the name of a data type present in the working copy. If it is, the next (and succeeding) fields (until a new data type name is found) are used to replace the current value for the specified column(s) of the relation in each of the next n rows.

Default:

If n is omitted, it is assumed to be 1.

"e" request (execute Multics command)
"E" request (execute Multics command)

Format:

EMultics command line
eMultics command line

Purpose:

Pass Multics command line to the Multics command processor for execution .

Example:

The request

```
Fhmu;df >udd>e>apl>test.doc;rdy
```

would be passed to the Multics command processor. The Multics command processor would execute the commands

```
hmu
df >udd>e>apl>test.doc
rdy
```

responding with a message of the form

```
Multics 16.0xb, load 16.0/41.0: 15 users
r 1242 45.456 3106+1935
```

Index:

Unchanged.

Note:

If the "E" format is used, a message announcing the current mode of eds will be printed. Otherwise, no reply will be made by eds.

"f" request (find)

A. Format:

f/data-type/string-1/

Purpose:

Find the next occurance of the character string string-1 within the column specified by data-type.

B. Format:

f/data-type/

Purpose:

Establish a default search column for the find request. After this request has been issued, any find request given without a data-type will search through this column.
This format takes precedence over format C.

C. Format:

f/string-1/

Purpose:

Find string-1 within the default search column. Initially, the default search column will be the leftmost column of the relation. The B. Format sets a new default for a search column.

If string-1 is the name of a data type in the working copy then format D must be used, otherwise, eds will assume a new default search column (format B) is being specified.

D. Format:

f/data-type/string-1/.../string-n/

Purpose:

When given more than one string-i, the find request will attempt to find a line matching the "tuple" of string-I"s provided. If a string-i is the "*" (asterisk), then it will match with any string in that column. The "*" acts as a "wild-card" and matches anything.

Example:

f/Canning, H. F./*/4367/

would find the next occurence of the name "Canning, H. F." with any address (the wild card) and with the extenstion "4367". Also,

f/Brown, J. G./23-334/

would find the first tuple in the relation with name "Brown, J. G.", room "23-334", and any extension. This assumes that the default find column was set to "name".

Notes:

1) When a data-type is not given, the default search column is used. On entry into eds, this will be the left-most column of the relation.

2) The find request normally assumes that you have given only a left-most portion of the strings you wish to find. To suppress this feature issue the request with only the argument "-off" Datum successor "on" means that you may attempt to find a datum by specifying only the leftmost portion of it. If eds finds that a string-i is not an element of the data type for the column string-i is to be found in then eds will use the

datum alpabetically following string-1 in the data type.
For example:

f-on
f/name/Canning/

would (assuming that the relation phone_book is being edited) respond with:

! Canning, H. F. ! 23-335 ! 4367 !

However, in cases where there may be multiple names, such as with "Smith, J. T." and "Smith, A. R." no guarantee of locating the desired name is given. The use of datum successor is "on" upon entry into eds. If datum successor is "off" and a string-1 is not a valid datum then the find request will fail.

"F" request (format)

Format:

F or F/data-type/dfm/delimiter/

Purpose:

Control the formatting and printing of lines on the console. The dfm is the name of a data-format-module (dfm) which will be called using the string to be printed as an argument. delimiter is the delimiter string for output desired to be printed after the datum. (That is, to separate this datum from those which follow it.)

Defaults:

If the dfm is the null string, then no dfm will be called. Upon entry into eds, the defaults are dfm=none and delimiter=" ! ".

Notes:

The F request accepts dfm's in the form of program-name$entry-point. If no arguments are given, all formats are restored to the default value. If no dollar sign ("$") appears in the dfm field, then the program-name is assumed to be the entry-point name. If no such program-name can be found then a default program-name of "dfms_" is used.

Example:

F/name/names/<tab>/

is equivalent to:

F/name/dfms_$dfm_names_/<tab>/

"g" request (get requests from file)

Format:

g file-name

Purpose:

Gets editor requests from an ascii text file specified by the relative or absolute pathname "file-name".

Note:

If the entryname portion of file-name does not end in the suffix ".eds" then this suffix will be appended. Thus files to be read by eds must be named with a name ending in ".eds". For example, "g write" and "g write.eds" are equivalent.

G requests are recursive, thus an ascii text file being read by eds may issue a g request to read from a different file. When this second file is exhausted, eds will continue reading from the first file. When the first file is exhausted, eds will return to reading input lines as it did when first invoked. Normally, this will be the user's console, but if eds is invoked from an exec_com file using the "&attach" exec_com command then input will be read from the exec_com file. When eds is reading from a file specified by a g request, it is in g-mode otherwise, it is in non-g-mode.

When the program interrupt feature of eds is used, eds reverts to non-g-mode prior to entering edit mode.

Example:

It is possible to have files which perform a series of requests. Suppose that every time you write a relation, you want to sort it and print the number of lines in the file. Typing,

```
g write
```

where "write" is the name of a segment containing the following lines

```
S
w
,
g
```

will accomplish this. The final line consisting of a g returns eds to reading from the file (or console) prior to the request "g write". This line is assumed if the last line of a file is not "g".

"I" request (insert)

Format:

I/string-1/...../string-n/

Purpose:

Append the line specified by the string-i's to the bottom of the current working copy of the relation. If string-i is the null string, a "null-string" will be inserted into the relation in the corresponding column of the relation. The strings must appear in the "order" currently defined through the "O" request. If there are more columns in the relation than there are input strings, null strings or default insertion strings (see the "I" request) will be inserted into the remaining columns.

Index:

Unchanged.

Note:

The "length" of the current working copy will be increased by one if the insertion is performed. The inserted line will always be the last line of the working copy.

"I" request (ignore)

Format:

I/data-type-1/string-1/.../data-type-n/string-n/

Purpose:

Allow the data-types given to be Ignored by the change, replace, insert, find and locate requests. The data-types will continue to be printed unless killed using the kill request. Upon entry to eds, no data-types are "Ignored". In insert, duplicate and duplicate-and-delete requests, the associated string-i's (the default insertion strings) will be used to create lines.

Note:

The string-i's must be elements of the data types data-type-i. If not, the I request will fail.

Example:

Working in the relation sal_list, which contains the data-types name, term, percent_time, salary_type and salary, if we are only dealing with information for the FIRST TERM then the command

```
I/term/FIRST TERM/
```

would allow for the following:

```
i/Smith, H. F./50/map/18500/
bp
! Smith, H. F. ! FIRST TERM ! 50 ! map ! 18500 !
```

"k" request (kill printing)

Format:

k/data-type-1/. . ./data-type-n/ or k-all or k

Purpose:

Suppress printing of the data-types specified. If specified with no arguments, all printing is suppressed. This does not affect the "p" request. When provided with arguments, the specified data-types will not be printed by any eds request, including the "p" request, which would normally print out a line or lines. If -all is specified then all data-types are killed (and therefore all printing is suppressed).

Default:

Upon entry into eds, no data-types are killed.

Note:

When used with no arguments, the verbose-killed mode of each data-type is not modified.

"l" request (locate)

A. Format:

l/data-type/string/

Purpose:

Locate the next occurance of string in the column specified by the data-type data-type and set the line index to that line.

B. Format:

l/data-type/

Purpose:

If only the data-type is given, the default locate column will be set to the column specified by data-type, and all future uses of the locate request will search through this column unless overridden. The default search column may be overridden by merely specifying the data-type and the string to be searched for.

C. Format:

l/string/

Purpose:

This locates string within the default locate column. See Format B for further details.

Note:

If verbose mode is on, and the request is successful, then the new current line will be printed. Note that the default locate column will be used in searches unless overridden by including a data-type as the first argument to the request. Upon entry into eds, the default locate column is the leftmost column of the

relation.

Index:

If the request is successful, the index will be set to the line where the requested datum next occurs; if unsuccessful, the index remains unchanged.

Note:

The "l" request should be used only when the "f" request cannot pinpoint the desired string; "f" is much more efficient than "l"

"m" request (merge or union)

Format:

m relation-name

Purpose:

Merge (or union) relation-name with the current working copy. relation-name may be either the name or reference number of a quart or relation.

Note:

See the description of union elsewhere in this guide for a complete explanation of this operation.

Index:

The line index will be set to the first line of the resulting relation.

Spacing:

A space may appear between the request and the name to be merged.

"M" request (meter editor use)

Format:

M-on or M-off

Purpose:

Allows programmers to meter eds's efficiency in order to improve operation. See documentation of the commands "eds_use" and "zero_usage".

"n" request (next)

Format:

nm or nmp

Purpose:

Set the line index to m lines from the current line index. If the request terminates in the letter "p", the new current line will be printed. If the integer m would cause the next line to be greater than the current length of the relation, the message "EOR" (End Of Relation,) will be printed on the console and the

current line index will be set to the last line of the relation.

Spacing:

Any number of spaces may appear between the request and m. An arbitrary number of spaces may also appear between m and p.

Default:

If the integer m is absent, it is assumed to be 1.

"0" request (order)

Format:

0 or 0/data-type-1/..../data-type-n/

Purpose:

If the arguments data-type-i appear, then the apparent order of the relation being edited is changed to match the order in which the data-type-i's were typed in. If no arguments are given, the order is restored to its default order, that being the current sorting order for the relation.

Note:

This command does not modify the current sorting order of the relation currently being edited. The only thing modified is the order in which the columns of the relation are read from and printed on the console. Note also that you need not specify the full range of data-types present. eds will append the extra data-types to the end of the request line and inform you that it has done so by printing those data-types which were not given.

Example:

In the relation phone_book, the normal sorting order is <name room extn>. The request:

0/room/extn/name/

would result in all lines printed after the request being printed in <room extn name> order. Also, all lines typed in using input-mode, replace, insert, ", or D would be interpreted as being in <room extn name> order.

"p" request (print)

Format:

p or pn

Purpose:

Print out n lines of the relation beginning with the

current line, then set the line index to the last line printed. Only those data-types in verbose mode (see the v request) are printed.

Default:

If the integer n is missing, it is assumed to be 1.

Spacing:

An arbitrary number of spaces may appear between the request and n.

"p" request (project)

Format:

P or P/data-type-1/. . ./data-type-n/

Purpose:

Project the current working copy on the specified data types. If no data types are specified then the current sort order of the working copy is assumed. If data types are present then the apparent order of the columns is reset to the projected order of the columns. Any previous order set by a "O" request is overridden.

Note:

This operation may produce an unsorted relation with duplicate rows. See the description of project for an explanation of the project operation.

Index:

The line index is set to the first line of the resulting relation.

"q" request (quit)

Format:

q -options-list- or q options

Purpose:

Quit from eds. The options may be chosen from among the following; if more than one option is desired, the letters should be concatenated.

d	delete the working copy
S	sort the working copy
w	write the working copy
f	force exit from eds

Note that the "f" option overrides or ignores the "S" and "w" options. When you type "q", eds checks to see if you have sorted the working copy and written it. If not, a message is printed on the console and you are asked to type "qf" if you wish to quit. After this message, the user is still able to sort, write, find,

etc. and then quit.

Notes:

Note that the order in which options appear is unimportant. Sort is always done first, followed by write, followed by delete, unless the "f" option appears on the line. The "f" option will cause all options except "d" to be ignored.

"Q" request (make quart)

Format:

Q/data-type-1/..../data-type-n/

Purpose:

Create a quart with the data-types specified by data-type-1 through data-type-n. The quart will contain a single tuple, consisting of the null element for each data type specified. The reference number of the resulting quart will be printed.

Index:

unchanged.

"r" request (replace)

Format:

r/string-1/...../string-n/

Purpose:

Replace the current line with the line specified by the strings string-i. If string-i is the null string, the i'th column remains unchanged. If there are fewer string-i's than columns, the remaining columns will be unchanged.

Index:

Unchanged.

"R" request (read a relation)

Format:

R relation-name

Purpose:

Reads the relation specified by relation-name into the current working copy. At the same time it changes the name of the current working copy to eds.relation-name and changes the relation being edited to relation-name.

Index:

The line index is set to the first line of the new working copy.

Note:

The previous contents of the working copy are destroyed.

"S" request (sort)

Format:

S or S/data-type-1/...../data-type-n/

Purpose:

Sort the working copy of the relation being edited. If no arguments are given, the copy is sorted by its current sorting order, not in the "order" specified by any prior "O" request having explicit data-types associated with it. If arguments appear, they must be separated by break characters. The relation will then be sorted in the order of the given arguments. If fewer arguments are given than there are columns in the relation, then those columns not provided will not be sorted. If arguments appear then the apparent order of the columns is reset to the new sorting order of the working copy. Any previous order set by a "O" request is overridden.

Note:

For more detailed information on Sort, see the description of sort elsewhere in this guide. The S request also removes duplicate rows from the relation by using the project operation on the sorted copy.

Index:

The line index will be set to the first line of the resulting relation.

"t" request (top)

Format:

t or tp

Purpose:

Reset the line index to the top of the current working copy, i.e., set the line index to one. If "p" is present, the top line of the working copy, the new current line, is printed.

"updelete" request (updelete)

Format:

updelete

Purpose:

Delete all lines above the current line. As a safety measure, the complete string "updelete" must be typed.

Notes:

The current line is not deleted by updelete.

"U" request (use)

Format:

U or U/data-type-1/...../data-type-n/

Purpose:

Use the given data-types. If no arguments are given, all data-types are restored to being used (rather than being Ignored). Upon entry to eds all data-types are "Used".

Defaults:

If no arguments are given, then all data-types will be used.

Note:

This request is the opposite of the "I" request.

"v" request (verbose printing)

Format:

v/data-type-1/. . ./data-type-n/ or v-all

Purpose:

Cause eds requests which print lines of the relation on the console to print the datums associated with the data-types specified. If -all is specified then all data-types will be printed.

Default:

If no arguments are given, the v (verbose) request turns on printing for those data-types not explicitly killed by a "k" request.

Note:

The "v" request is the opposite of the "k" request. A data-type is never printed if it is killed. (A data-type that is not killed is said to be in verbose mode.)

"w" request (write)

Format:

w or w relation-name

Purpose:

Write (save) the current working copy of the relation being edited under the name relation-name if relation-name is given. If not, the working copy is written with the name of the relation being edited, thus replacing the original version of the relation with the current working copy.

Note:

WARNING: In order to save the editing that has been done, a write request should be issued before exiting from eds. An optional space may appear between the request and relation-name. When the write is performed, a message is printed on the console informing the user of the date and time the relation was written.

"X" request (cartesian product)

Format:

X relation-name

Purpose:

Perform the cartesian product of the current working copy with the relation specified by the relation relation-name. relation-name may be either the name or reference number of a quart or relation.

Note:

See the description of cartesian product (cart_prod) for details of the cart_prod operation.

Spacing:

A space may appear between the request and relation-name.

Index:

The line index will be set to the first line of the resulting relation.

"-" request (minus)

Format:

-n or -np

Purpose:

Move the index "backwards" (towards the top of the relation) the number of lines specified by the integer n. If the new line index would be less than one, the message "NO LINE" is printed and the line index is set to 1. If the request is terminated with the character "p", then the new current line is printed on the console. Otherwise, nothing is printed.

Spacing:

Any number of spaces between the request and n may be present. An arbitrary number of spaces may also appear between n and p.

Default:

If n is absent, it is assumed to be 1.

"=" request (line number)
Format:

```
=
```

Purpose:

Print the current value of the line index. If an argument appears following the "=", the current value of the index is saved (for later use) instead of being printed.

Example:

The request

```
=x
```

causes the current line index to be saved. Then following the execution of additional requests, such as

```
n10p
r/.......
n
d etc.,
```

the index can be restored to its original value (i.e., its value when the request "=x" was issued) by using the "a" request and typing "a=".

Index:

Unchanged by the "=" request.

"," request (size)
Format:

```
,
```

Purpose:

Print the value of the current line index, as well as the current length and order of the working copy of the relation.

"~" request (abbreviation)
Format:

```
~sms_abbrev_line
```

Purpose:

Pass sms_abbrev_line to sms_abbrev. If sms_abbrev_line is the string "-on" or "-off" the control of the abbreviation and expansion feature is set to "on" or "off". If expansion is "on", anything between break characters will be examined for abbreviations, and those abbreviations found will be expanded and used in

the requests. If not "-off" or "-on", then the request must be one of the following:

COMMAND	PURPOSE	FORMAT
a	define abbrev	a abbrev expanded
d	delete abbrev	d abbrev
u	establish profile	u pathname
p	print profile name	p
l	list abbrevs	l

Note:

On entry to eds, expansion of abbreviations is "on". See description of sms_abbrev elsewhere in this guide for further details.

"<" request (verify input - protection)

Format:

< or </data-type-1/..../data-type-n/

Purpose:

Prevents the insertion into the data-base and into the relation of misspellings or incorrect data. This request requires that anything inserted, replaced, or changed in the given data-types exist within the data-type in which it is being placed (that is, already have a valid reference number associated with its character string representation). This request is of particular value for table data-types, as no entries should be inserted into such a data-type after it is created. Note that virtual data-types (such as those managed by dsm_integer, dsm_date, and dsm_decimal) cannot be protected in this manner, and any failure in insertion of a datum into the relation is due to improper typing of the datum.

If a misspelling is given to a "protected" data-type (i.e., if a datum is given which does not already have a reference number associated with it), a message is printed on the console informing the user that the given string could not be found in the associated data-type and asking if it should be inserted. A response of "yes" or "no" is required. If the response is anything but "yes" or "no", the question is repeated until a "yes" or "no" answer is provided.

Example:

(user typing is underlined)

<u></u>

1/Brown, J. &./3-403/7958/
eds: name protected. Insert 'Brown, J. &.'?
no
eds: replace by Brown, J. D.
bp
1 Brown, J. D. ! 3-403 ! 7958 !

If you decide that it would be easier to retype the original request rather than retyping the datum, you can answer the question
eds: replace by
with the mode-change character. eds will remain in its current mode, but the line it was in the process of adding will be ignored.

Defaults:
If given with no arguments, all data-types will be protected. When eds is entered, all data-types are protected by default.

">" request (do not verify input)

Format:
> or >/data-type-1/...../data-type-n/

Purpose:
Release protection on the data-types given.

Defaults:
If no arguments are present, then all data-types are un-protected.

Note:
This request is the opposite of the "<" request.

" request (duplicate)

Format:
"n/data-type-1/string-1/..../data-type-m/string-m/
or
"n/data-type/string-1/string-2/...../string-m/

Purpose:
This command duplicates the n lines beginning with the line specified by the current line index, placing them at the bottom of the relation. The strings of the data-types specified by the data-type-i's are replaced with the corresponding string-i's. In the second format , the data-type specifies the column in which replacement is to begin. From that column on (or until another data-type-i is found), the request functions similarly to insert. It is possible to mix the first and second formats together. If verbose mode is on,

the newly added line will be printed.

Example:

In rank_list, sorted by name-rank-title-soc_sec_num

"/title/Professor Emeritus/203380252/

would take the name and rank from the current line and insert a new line at the bottom of the file with the title "Professor Emeritus" and the soc_sec_num of "203380252". The request

"/title/Professor Emeritus/soc_sec_num/203380252/

is equivalent to the first example.

Index:

The current line index is unchanged. If n is absent, 1 is assumed.

Note:

See the "D" request for ditto and delete operations.

"." request (change mode)

Format:

.

Purpose:

Change the input, edit, or increment mode of eds. In input mode, lines typed on the console are inserted at the bottom of the relation, much as in the insert (i) request. For example, in input mode, typing

/Brown, J. D./3-403/7958/

would be equivalent to typing the following in edit mode.

i/Brown, J. D./3-403/7958/

From edit mode, typing "." followed by a carriage return permits you to enter input mode. From input mode, typing "." followed by a carriage return will return you to edit mode. While in input or increment mode, it is possible to execute any request available in edit mode by simply typing a "." followed by the request. For example,

.S/name/extn/room/

would allow you to resort the relation while in input or increment mode.

In increment mode, typing "." returns you to the mode (INPUT or EDIT) from which increment was entered.

Index:

While in input mode, the current line index is set to the last line inserted. Upon a normal return to edit mode (via the mode-change character), the line index is restored to the value it had upon entry into input mode. The contents of the line specified by this value for the line index may not be the same as when input mode was entered due to editor requests issued while in input mode. If edit mode is reentered by using the program interrupt feature of eds the line index will specify the last line of the relation.

"?" request (status)

Format:

? or ?request

Purpose:

Provide information on the internal status of eds for the "?" request including:

1) Multics-SMS name of the relation being edited and the name of the working copy.

2) printing status, whether a write has been issued, whether a sort is necessary, if datum successor will be used in the find request (this means that a datum may be found by specifying only the left-most section: see (3) in the "f" request below), and whether sms_abbrev will be used to expand abbreviations.

or

Provide information available concerning the status of the given request, where request is one of the following: 0 < > f l I U F k v S g

?0	current order of data-types
?<	current data-types protected
?>	data-types unprotected
?f	default find column
?l	default locate column

?I	data-types ignored and default string
?U	data-types used
?k	data-types killed
?v	data-types verbose
?F	formats for data-types
?S	present sort order of editing copy
?g	recursion depth and list of files

"_" request (difference)

Format:

_ relation-name

Purpose:

Take the set-theoretic difference between the current working copy and relation-name. relation-name may be the name or reference number of a quart or relation.

Spacing:

A space may appear between the request and the relation name.

Index:

The line index will be set to the first line of the resulting relation.

"&" request (intersect)

Format:

& relation-name

Purpose:

Intersection of the working copy and relation-name. relation-name may be the name or reference number of a quart or relation.

Spacing:

A space may appear between the request and the relation name.

Note:

See the description of intersection for details of operation.

Index:

The line index will be set to the first line of the resulting relation.

"+" request (increment and repeat)

Format:

+nrequestn or +n<request><arguments>n

Purpose:

This request permits the iteration of any eds request. While in incremental mode, the request being performed

is specified by <request>, and may be any valid eds request. (Caveat Emptor) The integer n specifies the "increment" of the current line index after each request. It is possible to use this request to modify only every n-th line in a relation.

The <arguments> are those which would normally appear with the <request>. If <arguments> is not present, they will be "read" from the console for each iteration of the request. This allows increment mode to work like input mode. Issuing

```
+1l
```

is equivalent to typing

```
.
```

with the exception that the editor mode is announced as being "INCREMENT".

Example:

Assume a relation containing "name-term-salary" information. Also assume that each name has one entry for each of the three terms. To modify only the "FIRST TERM" salaries, the following requests could be issued

```
:-off
+3Dp
INCREMENT
! Miller, R. J. ! FIRST TERM ! /salary/1800/
! Bruce, J. D. ! FIRST TERM ! /salary/1000/
```

Note:

If the string "p" is present, the new current line will be printed each time the line index is incremented, prior to execution of <request>. To exit from increment mode, type the mode-change character ".". In order to execute any other eds request, simply begin the request with the mode-change character as you would to execute an eds request from input mode. To skip a particular line of the relation, type an empty line.

":" request (suppress carriage returns)

Format:

:-on or :-off

Purpose:

Suppress (or include) carriage returns at the end of each line of the relation.

Default:

On entry to eds, carriage returns are on.

SOME WARNINGS AND NOTES ON USING eds

1. Insertion and replacement of lines may cause the sorting order to be disturbed. If this should become inconvienient during editing, sort the file using the "S" request.

2. eds has a Multics pi (program_interrupt) handler to allow you to quit in the middle of a request, terminate the request and have eds await a new request. This is accomplished by hitting the attn (attention) button, waiting for the system to respond with "QUIT", and then typing "pi". eds will then return to "EDIT" mode and await the next editing request.

3. Since the use of the the locate request is relatively slow, it is advisable to use the find request wherever possible. The multiple find over more than one data-type is quite fast and efficient. It also saves the user the trouble of typing several requests to get to a specific line in the relation.

4. It is less expensive to use the "D" request to make a change to a line than to use the "c" request. Therefore, for "cheaper" editing as well as faster response, use the "D" request wherever possible.

5. Notice that a misspelling which has been inserted will override "protection" mode operations within a given data-type. Thus, that same misspelling may continue to be inserted.

6. Note that in all examples and formats, the "/" (slash) has been used as a break-character. There is no fixed break-character. Any character not appearing within one of the arguments on the line may be used as a break-character for that line. The only restrictions are that the character be a non-numeric, printing ascii character and that the character be the only character used as a break-character on that line. As an example, the lines:

```
f/name/Brown, I. M./
f$name$Brown, I. M.$
f!name!Brown, I. M.!
```

   are treated identically. Notice however, that a line of the form:

f.name.Brown, I. M..

will be handled incorrectly.

Also note that requests which require break-characters were shown with terminal break-characters. Unless the string being examined contains terminal blanks, the final break-character is not required. An example of this is:

f/name/Bruce, J. D.

where the terminal "/" has been omitted. The requests

F/rank/ ! /

and

F/rank/ !

are not equivalent.

To include terminating spaces (or tabs) in a request line, a final break character must be included. Since leading and trailing blanks in data-type names and datums are not ignored, they should be omitted unless specifically desired.

7. The "null string" is represented by the appearance of two break-characters immediately adjacent to one another. The string "//" is a null argument. Do not confuse a request that has null arguments and a request that has no arguments.

8. If expand mode is on (see the "-" request), any character string between two break-characters may be an abbreviation. An attempt will be made to expand it. Abbreviations should be chosen with care, but they can never occur as parts of datums. The expansion of a character string between break characters can be inhibited by prefixing it with \c (backslash c or cent c on IBM2741 type terminals). Thus if FT is an abbreviation for FIRST TERM then /FT/ is equivalent to /FIRST TERM/; however, /\cFT/ is taken as the character string /FT/.

9. The requests C, m, P, Q, X, _, and &, which provide access to the RDMS relation operations, accept any format acceptable to the command make_quart in the specification of a relation. Thus if a quart with the term FIRST TERM is desired, the request:

Q/term=FIRST TERM/

will create such a quart.

SOME WARNINGS AND NOTES ON USING eds

DEFAULTS TO eds REQUESTS

Requests with no arguments:

request	equivalent
-	-1
d	d1
a	a1
p	p1
n	n1

request	default
<	protect all
>	release all
U	use all
S	sort using current sort order
P	project using current sort order
O	set printing order to correspond to current sort order
k	stop all printing except for p request.
v	enable printing of data-types in verbose mode
g	read from previous input file or user's console.

Requests without any data-types:

request	result
f/string-1/.../string-n/	use the default search column
l/string-1/	use the default search column
c/string-1/string-2/	change all occurences of string-1 to string-2 in the currrent line.
"/string-1/.../string-n/ D/string-1/..../string-n/	assume leftmost column for string-1.

INITIAL EDITOR STATUS

Upon entry to eds the editor's status is:

1. Expansion of abbreviations is "on".
2. Default locate and find columns are the left-most columns of the relation.
3. The order of the relation is the real sorting order.
4. All data-types are verbose (i.e., printed).
5. Datum successor (see the find request) is "on".
6. All data-types are protected.
7. All data-types are used.

(END)

Command
09/24/74

Name: evaluate

Evaluate is the command level interface to the relational operator. It is invoked with the names of the input relations to the relation operator and the name to place on the result. Some of the arguments (as specified below) can be strings (referred to as specifiers) acceptable to the make_quart (mqrt) command. The relation operator is invoked with the reference numbers of the input relations, and evaluate names the resultant relation with the name specified by the user. relation operator.

Usage

evaluate input output_map rop_def arg resultant

1) input — is the name of the input relation.

2) output_map — is the name or specifier of the output map relation. This relation specifies which of the input values and computed values are to be placed in the resultant relation. It also specifies the order of the resultant relation.

3) rop_def — is the name of the relation operator definition relation. This relation specifies the computation to be performed by the relation operator.

4) arg — is the name or specifier of the arg relation. This relation provides certain constant values to which a rop definition may refer. The argument relation must be provided, although it may be a null relation (cardinality and arity of zero).

5) resultant — is the name to be placed on the resultant relation. This relation will have a sort order as specified by the output_map relation, although it is not necessarily a sorted relation. The result will replace any existing relation named resultant.

Example

```
evaluate dir /name/num_names/ count // dir.count
```

The rop definition relation "count" would be used to compute an algorithm on the input relation "dir" with the output order consisting of 2 data types, "name" and "num_names", producing the resultant relation named "dir.count" The output_map and argument relation arguments are specifier strings which are converted to relations by evaluate. In this example, the argument relation is null.

Notes:

If the sort order of the result does not match the sort order of the input relation then the result relation may not be in sort and the sort command (part of sms_interface) must be used.

(END)

file_where

Command/Active-Function
09/24/74

Name: file_where
fwh

Using the "file_search_rules" currently in effect, file_where attempts to find the file named. If successful, the pathname is returned.

Usage:

file_where name1 . . . nameN
or
fwh name1 . . . nameN

For each namei supplied on the command line, file_where attempts to find a segment of that name using the current file_search_rules. If no segment is found an error message is printed on the console. Otherwise, if 1) file_where was invoked as a command the pathname of the segment is printed on the console, or 2) if file_where was invoked as an active function, the pathname is appended to the return string. (If more than one pathname is to be put into the return string they are separated by blanks).

Example:

1) fwh createdb.ec

Reply: >udd>RDMS>service.ec>createdb.ec

2) ioa_ [fwh createdb.ec]

Result: >udd>RDMS>service.ec>createdb.ec

Notes:

See documentation of "set_file_search_rules" and "print_file_search_rules".

(END)

Command
09/24/74

Names:

get_name_sets
gns
get_refno_sets
grs

This command provides a way to obtain the correspondence between a reference number and a set-name or vice versa. Given a reference number it prints the name of the set, or given a set-name it prints its reference number.

Usage:

```
get_refno_set  set1 set2 set3 . . . setI
     or
grs set1 set2 set3 . . . setI
     or
get_name_set set1 set2 set3 . . . setI
     or
gns set1 set2 set3 . . . setI
```

Where setI is either the reference number or the name of a set.

Example:

```
grs phone_book  acad_assign 50  feh
```

prints out:

```
feh is not a set
150 phone_book
154 acad_assign
 50 name
```

Note:

If setI is not the reference number or name of a data-type, relation, strategy module, or system table, an error message will be printed on the user's terminal.

(END)

Names: help

The RDMS version of the Multics "help" command assists users in obtaining information about RDMS or Multics. It differs from the Multics version of the command in that it normally searches for both RDMS and Multics "help files".

Usage:

help name1 -ctl_arg1 ... name2 -ctl_arg2

The parameters "name1", "name2", etc., specify "help files" which are to be examined by the user; for example, the command:

help print_set list_data_type eds_changes

would print portions of those files describing the "print_set" and "list_data_type" commands, as well as the file documenting recent changes to the relational editor "eds".

The "-ctl_arg1" (control arguments) are taken from the list below. None, one or more "-ctl_arg1" can appear anywhere on the line. The exact effect depends on the control argument.

-multics
-mul

Normally the RDMS help command searches the directory containing RDMS help files before searching the directories containing Multics help files. If a Multics help file has the same name as an RDMS help file (for example, each directory contains an "motd.info" with a message of the day), it may be desirable to alter the order of the search so that the desired help file is found. If "-multics" or "-mul" appears on the command line, for all name1 following this control argument the multics directories are seached before the RDMS directory.

-rdms

If this control argument appears on the command line, for all name*i* following, the RDMS help directory is searched before the Multics help directories. For example,

"help motd -mul motd -rdms news"

will cause 1) the RDMS "motd.info" help file to be located and portions printed on the console, then 2) the Multics "motd.info" file to be located and printed, and finally 3) the RDMS "news.info" file to be located and printed (presumably there is also a Multics help file of the same name).

-pathname *path*
-pn *path*

Rather than searching the various directories for the help file, use the segment specified by *path*. (*path* may be either an absolute pathname (">udd>MITHFS>info_segs>x" meaning x.info in the MITHFS project's help directory) or a relative pathname ("x" meaning x.info in the working directory). As with all name*i*, if the name does not end in ".info", the suffix is appended.)

Notes:

For each name*i* the help command attempts to locate a help file name*i*.info, possibly searching various directories as described above. If the help file is not located a message "No help file name*i*.info" is printed on the console. Otherwise, help prints the contents of the file from the beginning to just before the first "control character" or to the end of the file if there is no control character. (The control character is a non-printing character with octal value 006. This character is not printed on the console.) If help has reached the end of the file, it prints a line consisting of "(END)" and is finished with the current name*i*. Otherwise help counts the number of lines from the current control character to the next control character or the

end of the file, and asks:

> xx lines follow,
> more help?

The user is required to answer "yes" or "no" (if any other response is made, help complains and tells the user "Please answer 'yes' or 'no'"). If the answer is yes the next group of lines is printed, otherwise help moves on to the next name1 (if any).

The control character may be placed in a file by typing "backslash 006" (that is, a "backslash" character immediately followed by the three digits 006, without intervening spaces) on an ascii terminal. Or if the terminal is non-ascii (if there is no "backslash" character the terminal is non-ascii) a different character must be used for the backslash: for example, on a selectric console such as a 2741, "cent-sign 006" may be used.

The version of the "check_info_segs" commands availible to RDMS users can be useful in discovering which RDMS or Multics ".info" segments have been modified recently. Thus a user may become aware that new info files are available or old ones have been changed if he types "cis" at regular intervals (perhaps every day). See documentation of "check_info_segs" for more information on using that command.

(END)

Names:

hmd

This command counts the number of users who have initiated data bases. The number is printed on the console.

Usage:

hmd -user-list- -ctl-args-

where:

-user-list- is a list of users that will be counted if they are logged in and have initiated a data base. If no -user-list- is specified, all users who have initiated data bases will be counted. The -user-list- consists of terms such as "McGary.RDMS" (count McGary if he is logged in under the project RDMS and has a data base set), "McGary" (count McGary logged in under any project if he has a data base set) or ".RDMS" (count any user of the RDMS project who has a data base set).

-ctl-args- is selected from the following list:

-abs which means that only absentee users will be listed. The default is to list all users.

-p <path> means that instead of checking the standard RDMS "whotable" to determine which users are logged in, the segment specified by <path> will be used as the "whotable". This is primarily a debugging aid.

Example:

1) If the user types:

hmd

then the following might be printed in response:

2 data bases are initiated.

If no data bases are set, the message is:

No data bases are initiated.

2) The command line:

hmd Caloggero .RDMS

might respond with:

2 data bases are initiated.

If Caloggero.EEAdmin and Goldman.RDMS were logged in and had initiated data bases, while McGary.RDMS was also logged in but did not have a data base set.

3) Finally,

hmd .RDMS .EEAdmin -a

would count the number of users on either project logged in "absentee" with data-bases initated.

Note:

See also the commands "hmp", "whop" and "whod".

(END)

Names: hmp

This command counts the number of people using the Relational Data Management System and prints the number on the console.

Usage:

hmp -user-list- -ctl-args-

-user-list-
is a list of terms containing user names and projects. A term such as "McGary.RDMS" causes the user McGary to be counted if he is logged in under the RDMS project, while a term such as "McGary" means count McGary logged in under any project. The term ".RDMS" would mean count all users logged in under the RDMS project.

-ctl-args-
is optional. If it occurs it is one or more of:

-a
Count only absentee users.

-d
Count only uses who have data bases set.

-p <path>
Use the "whotable" specified by <path> when counting. By default, the RDMS whotable is used. An entry is automatically added to this table whenever a data base is set. Additionally, some users are added to the table when they log in. This argument is primarily a debugging aid, so the normal user will have no occasion to make use of it.

Example:

```
hmp
```

might cause the following to be printed on the console:

```
(3 users)
```

Notes:

See the related commands, whop, whod and hmd.

(END)

insert_module

Command
09/24/74

Names: insert_module
inm
delete_module
dlm
rename_module
rnm
rename_module_force
rnmf

These commands enable a user to insert, delete, or rename modules in their data base. RDMS system modules, data strategy modules or relational strategy modules may be inserted, deleted, or renamed. The first four letters of the module name are used to determine the type of module being manipulated. These commands only recognise modules whose names begin with:

SMS_	for SMS system modules.
dsm_	for data strategy modules.
rsm_	for relational strategy modules.

Entry: insert_module, inm

Usage:

insert_module *module-name*
or
inm *module-name*

The name of the module, *module-name*, is entered into SMS_segments and assigned a reference number (refno). If the module is a dsm, it can then be specified when creating a new data type. Or if the module is an rsm, it will be possible to create relations with the new relational strategy, or convert quarts to the new strategy. However, no attempt is made to locate a program named *module-name*.

If *module-name* is already used in SMS_segments, insert_module will print an error message.

Entry: delete_module, dlm

Usage:

delete_module module-name

or

dlm module-name

The information in SMS_segments associated with module-name will be deleted. (If there was no entry for module-name, the error is reported and nothing is modified.) The program named module-name, however, is not deleted. The program simply cannot be used from the currently set data base. Once a module has been removed from all data bases in which the program was known, the program may be deleted in a separate operation.

This entry can delete a strategy module from a data base even though a relation or data type managed by the module remains in the data base. It is the user's responsibility to convert the set to a new strategy before deleting the module (unless the data in the set is being discarded).

Modules are most often deleted when a newer version of the program is developed: for example, at one time dsm_v2_astring updated dsm_astring, so dsm_astring was deleted from several data bases after all data types had been converted. (See restructure_data_type for conversion between dsms, or sms_interface for conversion from quart to relation of a specific strategy and back.)

Entry: rename_module, rnm

Usage:

rename_module old-name new-name

or

rnm old-name new-name

In SMS_segments, old-name will be replaced by new-name. This will have the effect of renaming the module everywhere it is used in the data base. If new-name already appears in SMS_segments, the name will not be changed and an error messge is printed.

Entry: rename_module_force, rnmf

Usage:

rename_module old-name new-name

or

rnm old-name new-name

This entry works exactly like rename_module, except if new-name already exists in the data base it is first deleted so the name can be successfully changed.

Note:

When any of these commands executes successfully, the name and reference number of the module affected is printed on the console. (For rename set, this is old-name since that was the name manipulated. If the old-name was not already present in the data base an error was reported instead.)

(END)

Names: list_data_type, ldt

This command allows a user to print part or all of the contents of a data type.

Usage:

list_data_type *data_type* -ctl-args-

where the arguments to list_data_type are:

data_type
is the name or reference number of the data type to be listed.

-ctl-args-
is optional. If it appears, it includes one or more of the "control arguments" below:

-from *data element*
The datums will be listed starting at the first datum equal to *data element*, or if *data element* is not in the data type, the list will begin at the datum that logically follows it (i. e., the first datum lexicographically after *data element*). If no starting element is specified, the data type will be listed starting with the first datum.

-to *data-element*
The data type will be listed up to and including the datum specified, or if that datum is not in the data type, up to and including the datum that would follow it. If this control argument is not supplied, the data type will be listed through the last element.

-octal

In addition to character strings representing data elements, the corresponding reference numbers will be printed in octal.

-decimal

The refnos will be printed in decimal form. This is the default mode for reference numbers.

-norefnos

No reference numbers will be printed. Only character strings will appear. If this control argument does not appear, the reference numbers are printed.

-dfm dfm-name

The format in which the data elements are printed will be that of the data format module specified. See the overview of "dfms" and documentation of each dfm. By default, no dfm is used so the data elements are displayed in the "standard output form" of the managing data strategy module. (For example, while dsm_date accepts various input forms such as "02/27/74", "27feb74" or "February 27, 1974", the output form for all of these is "February 27, 1974". Thus a user wanting a briefer form of the date on output might use dfm_mmyy_ to get "2/74".)

Example:

ldt acct_num -from 79540

The data elements of acct_num will be listed starting at 79540. Since acct_num is a virtual data type managed by dsm_integer, account numbers will be printed up to the largest possible number, which would be an excessively long list. The person typing such a command probably intends to "quit" after seeing only part of the list.

ldt name -from M

The name data type will be listed starting at names beginning with M and ending with the last name in the data type.

If the reference number of the name data type were 75, then

```
ldt 75 -from M
```

would be equivalent to the previous command line.

```
ldt name -from J -to K -dfm abbrev_name
```

would list names starting with J as well as the first name starting with K. The names will be abbreviated (i. e., last name and one or two initials). This might be useful if a user needed to know how to spell someone's last name knowing it began with a J. In this case, the printing can be speeded up by omitting the full form of first and middle names. For a fairly small "name" data type, the resulting list might be:

```
James, A. K.
Jones, D.
Jufal, T. L.
Keasal, D. R.
```

Note that the last name was the first of the "K's".

Notes:

If -dfm is used, the dfm will be located according to the conventions for naming and locating data format modules. Thus, specifying either dfm_abbrev_name or abbrev_name as the data format module following the -dfm keyword is equivalent.

(END)

Names: list_sets, lss

The names and reference numbers of a set (relation or data type) or sets specified is printed on the console.

Usage:

list_sets set1 set2 . . . setN -ctl-args-
or
lss set1 set2 . . . setN -ctl-args-

1) set1 specifies the set to be listed, either by reference number or by name.

2) -ctl-args-
is optional. If it appears it is one or more sets of "control arguments" taken from the following list:

-relations
-rel — Relations will be listed.

-system — System sets and modules will be listed.

-dsms — Data strategy modules will be listed.

-rsms — Relational strategy modules will be listed.

-data-types
-dts
Data types will be listed.

-quarts — Quarts will be listed

-all — All sets (relations, data types and quarts), strategy modules (rsms and dsms) and system objects will be listed.

-interval start end
Objects with reference numbers in the range specified will be listed.

Example:

```
lss -rel -dts
```

will list relations and data types currently present in the data base.

```
lss old.* -rel -dts
```

would list all data types and relations with two component names whose first component was "old", then all relations would be listed, then all data types would be listed. (The control arguments "-rel" and "-dts" do not restrict the range of the star name "old.*".)

Notes:

The star convention may be used in the arguments set1 set2 . . . setN. See the Multics Programmers' Manual (MPM) for a discussion of the star convention.

See also the commands sms_star and status_set for obtaining other information concering the date-types and relations in a data base.

(END)

Command/Active-Funcion
09/24/74

Names: memory, mem

This program can be used as a command to set the value associated with a name, delete this association or list the value associated with a particular name, or list all current associations. When used as an active function, memory returns the value of a name, or if the name is not associated with any value, a special value indicating no association is returned.

Usage:

I. Use as a Command

memory control_argument

The control arguments meaningful for use in command mode are:

1) -set name value

Set the value of name to be value. If name is already associated with some value, the old association is deleted and the new established.

2) -list name
-ls name
-list
-ls

List the value of name, or if no name appeared, list all name-value associations.

3) -delete name
-dl name

Delete the association between name and its current value.

4) -pathname path
-pn path

Normally, memory stores the associations between names and values in a segment >udd>Project>User>User.memory_seg. If the user wishes to use a different memory seg, the -pathname control argument can be used. When memory is invoked the first time it creates >udd>Project>User>User.memory_seg and

likewise when instructed to use a different memory seg, it will create the other segment if it does not already exist. (Memory notifies the user whenever it creates a segment, or if unable to create a segment when it is necessary.)

5) -push name value

This control argument functions exactly like -set, except that if name has already been associated with a value the old associaton is saved, and can be retrieved using -pop in either command or active function mode.

6) -pop name

One value currently associated with name is deleted and if previous associations were "pushed" another is restored; otherwise name is undefined and if the value of name is desired the invalid indicator (see below) is used. Currently "popping" does not restore values in First-In-Last-Out order. It is planned to implement this at some point. The control argument "-pop" can still be useful for obtaining all values associated with a name, although in an unspecified order.

II. Use as an Active-Function.

[memory name invalid_indicator af_control_args]

name

The name whose associated value is to be returned.

invalid_indicator

(Optional.) If name does not have an associated value then memory returns the value of invalid_indicator. If invalid_indicator is omitted then the string "undefined!" is returned.

af_control_arg

(Optional.) If this control argument appears it may be one only of the following:

-pop
-delete
-d!

After the value is looked up, one value associated with the name is deleted. See "-pop" as described above as a command control argument.

-increment
-inc

If the value associated with name is integral, after lookup it will be incremented by one.

-decrement
-dec

If the vaue associated with name is integral it will be decremented by one after lookup.

Example:

Assuming that the following call to memory is the first invocation, the responses below can be expected. Initially there are no associations between names and values.

To set the value associated with a name:

```
memory -set x y
```

Since this is the first call to memory a memory segment is created:

```
memory: creating >udd>Project>User>User.memory_seg
```

The value "y" is now associated with the name "x".

```
memory -ls
```

The response will be:

```
x   y
```

At this point only the one name-value association exists in the memory segment.

To list a particular name's current value(s):

```
memory -list x
```

The response is again the same since only "x" has been given a value.

At this point only the name "x" is defined. Thus:

```
memory -list z
```

produces the repsonse:

```
z          undefined!
```

The following is an example of the active function use. (ioa_ is a command which prints its argument. The bracketed part of the command is expanded into the active function's return value before ioa_ is called.)

```
ioa_ [memory x]
```

The response is to print the current value of the name "x":

```
y
```

Active Function use with undefined name:

```
ioa_ [memory z]
```

"z" is undefined so the value printed is:

```
undefined!
```

Active Function with user supplied invalid indicator:

```
ioa_ [memory z test]
```

Here the user supplied "test" to be used as the invalid indicator:

```
test
```

Active Function with "-pop":

```
loa_ [memory x -pop]
```

Get the value associated with "x", but delete it after retreiving it:

```
y
```

Since the -pop causes one value associated with the name "x" to be deleted and only one association existed, "x" is now undefined:

```
memory -ls x
```

produces the response:

```
x          undefined!
```

Note:

The use of active functions is described in the Multics Programmers' Manual, Part II, Reference Guide to Multics, Section 1.4: "The Command Language".

(END)

Names: new_data_type, ndt

The new_data_type command allows a user to create a new data-type with a specified data strategy module. (A data type must be created before it can be used. When creating a new relation, it is often necessary to create any new data types that will be used in the relation. The relational editor "eds" can also be used to create a new data type.)

Usage:

new_data_type *data-type* *dsm-name* -ctl_arg-
or
ndt *data-type* *dsm-name* -ctl_arg-

where the arguments to the rdt command are:

data-type specifies the name of the data-type.

dsm-name is the name of the data strategy module which will manage the data-type.

-ctl_arg- is an optional argument which if present must be "-brief" or "-bf". If this argument is specified then the message indicating the reference number assigned to the new data type is suppressed.

Notes:

Currently available data strategy modules are:

dsm_integer	(virtual)
dsm_table	
dsm_v2_astring	
dsm_room	(virtual)
dsm_acct	(virtual)
dsm_subject	(virtual)
dsm_decimal	(virtual)
dsm_ciph_integer	

Virtual data strategy modules encode the character string datum directly into a 36 bit word and do not require additional storage for the character string datums. Non-virtual dsm's generate a reference number whose value is used to locate the character strings stored in a data type segment in the user's data base directory.

Certain data strategy modules (especially dsm's which are non-virtual dsm's) ask the user for additional information when a new data type is created. The questions may request the following information:

1) the maximum length allowed for a datum in the new data type (dsm_table, dsm_v2astring)

2) the maximum number of datums the data type may contain (dsm_table)

3) the pathname of the rotor archive for ciphering purposes (dsm_ciph_integer) (See the documentation for dsm_ciph_integer.)

4) the name to be used for the ciphering machine created when ciphering data for this data type or used when the message:

   name: password

   asking for the user's password is typed on his console (dsm_ciph_integer) (See documentation for dsm_ciph_integer.)

Example:

1) The command line:

ndt grade dsm_integer

creates a data type named "grade" managed by data strategy module "dsm_integer". new_data_type will inform the user that the data type has been created by typing:

New Data Type grade with refno 120 and strategy module dsm_integer.

Since all data strategy module names begin with "dsm_" new_data_type does not require the user to type this prefix. The above command could have been typed:

ndt grade integer

with the same results.

2) The command line:

ndt student_name v2_astring

would create a non-virtual data type named "student_name" and managed by dsm_v2_astring. Since dsm_v2_astring requires additional information regarding this new data type, the user is asked the following questions: (user input is underlined)

dsm_v2_astring: What is the maximum string length?
80

New Data Type student_name with refno 121 and strategy module dsm_v2_astring.

dsm_v2_astring (and any dsm imposing a maximum string length) will not accept a datum whose length is longer than the maximum string length. Thus the user is assuming that student names are never longer than 80 characters. The maximum string length is also important relative to data format modules. Since data format modules normally overwrite the character strings returned by a data strategy module by the get datum primitive, the maximum string length associated with a

data type must be large enough to contain the longest character string a data format module used to reformat datums of the data type might produce.

The non-virtual data strategy module "dsm_table" also asks for a maximum string length. In addition, it requires the maximum number of data elements the data type is to be allowed to contain. An attempt to insert a datum into a table data type which is full will fail.

(END)

print_data_base

Command
09/24/74

Name: print_data_base, pdb

This command provides the absolute pathname of the currently initiated data-base.

Usage:

print_data_base
or
pdb

(END)

Name: print_file, prf

The print_file command is used to print an ascii file a specified number of times. It is useful when a number of copies of a file (such as a list of labels or addresses) is needed.

Usage:

print_file file-name -copies-

file-name is the pathname of the file to be printed. If file-name is not a pathname (i.e., it is a entryname) then the file search rules will be used. Otherwise the directory specified by the pathname (it may be relative or absolute) is searched.

-copies- is the number of copies of file-name to be printed. If this argument is omitted then 1 copy of file-name is printed.

Example:

print_file labels.memo 5

prf notices.info

Note:

Print_file writes on the stream "user_output" so its output can be directed to a file using the file_output command.

(END)

print_file_search_rules

Command
09/24/74

Names: print_file_search_rules, pfst

The print_file_search_rules command causes the printing of the currently operating file search rules in the user output stream.

Usage:

print_file_search_rules

Example:

print_file_search_rules
or
pfsr

Notes:

See also set_file_search_rules.

(END)

Names: print_set, prs

This command permits the user to print on his console the contents of a relation. All or part of a relation may be printed with some control over format. The length (number of rows), order (number of data-types in relation), ent (list of data-type names in the sort order of the relation), and the type of strategy module for the relation are also printed.

Usage:

print_set relation arg1 arg2. . . argn

or

prs relation arg1 arg2. . . argn

where relation is the name or reference number for a relation and the argi are one of the following:

-brk string
-break string — When the relation is printed, each tuple (or row) of the relation is printed as a line on the console with each datum separated from the next datum by a string, called a break string. By default this break string is " | ", but the -break (-brk) argument can be used to change the value of the break string. The string argument following the -break (or -brk) argument is used for the break string instead of the default value.

-from number — Normally the printing of the contents of the relation starts with the first tuple. This argument can be used to start the printing on a different tuple. Printing begins with the tuple indicated by number. However, if number is less than or equal to zero, printing starts with the first tuple.

-to number — Normally printing continues to the last tuple of the relation. This can be changed by use of the -to argument. Printing continues up to and including the row indicated by number. If number is less than zero or greater than the number of tuples in the relation, printing continues up to

and including the last tuple of the relation.

-rows number — This argument allows a user to specify the number of tuples (or rows) he wants printed. If number is less than zero then all tuples after the starting tuple will be printed. If equal to zero then no tuples will be printed. (This is useful if only the sort order or size of the relation is desired.)

-octal — This argument causes the reference numbers and not the strings they refer to to be printed. The reference numbers are printed in octal.

-decimal — This argument is similar to the -octal argument, except that the reference numbers are printed in decimal. This argument takes precedence over the -octal argument.

-characters — This is the default printed mode for print_set. That is, the output is the strings or characters represented by the reference numbers stored in the relation. This argument takes precedence over the -octal or -decimal arguments.

-brief — This argument specifies that the header information (strategy module, length, order, and ent) should not be printed. The default is to print the header information.

Notes:

The last occurrance of an argument takes precedence over previous occurrances of the same argument. The arguments may apear in any order after the relation argument. The star convention may not be used.

(END)

Names: quick_report, qr

quick_report enables the user to print a single relation in the standard report format.

The program uses a relation and two segments, the "sort-order-file" and the "print-order-file", prepared by any text editor such as edm or qedx, and produces a file which is compatible with the Multics "runoff" program. The output segment will have a header for each column, with each column aligned on the page as specified by the user.

Usage of the sort-order-file and the print-order-file:

sort-order-file: This provides quick_report with information necessary to sort the relation as specified before printing. The format of the segment must be:

data-type1
data-type2
.
.
data-typen

A copy of the relation will be created and sorted in the order

data-type1/data-type2/...../data-typen

print-order-file: This provides quick_report with additional information needed to produce the final output form. The format of the segment is

<dlm> data-type1 <dlm> column number1 <dlm> header1 <dlm> dfm1
<dlm> data-type2 <dlm> column number2 <dlm> header2 <dlm> dfm2
.
.
<dlm> data-typen <dlm> column numbern <dlm> headern <dlm> dfmn

where <dlm> is a delimiter consisting of a single ASCII character, not otherwise used in the line. It may differ on each line. If the dfmi is not given for a row of the print-order-file, no delimiter should typed after the header in that row.

The output segment will then be arranged

header1	header2	...	headern
datum11	datum12	...	datum1n
datum21	datum22	...	datum2n
	.		
	.		
	.		
datumn1	datumn2	...	datumnn

with each datum aligned on the given column.

Usage:

quick_report -opt1 -opt2 -opt3 ... -optn

where the -optI are selected from the following list:

-rel_name name
-rel name

where name is the name of the relation from which the report is to be produced.

-sof name
-s name

where name is the segment name of the sort-order-file. The segment must have the suffix ".sof" as part of its name, though the suffix does not have to be included when typed on the command line. Either "name.sof" or "name" will work correctly. If no sort-order-file is found, the data-types listed in the first column of the print-order-file will act as the sort-order-file.

-pof name
-po name

where name is the segment-name of the print-order-file. The segment must have the suffix ".pof" as part of its name, although the suffix does not have to be included when typed in the command line. Either "name.pof" or "name" will work correctly. In the pof, the dfmI specification is optional, but all the other

information is required. Quick_report uses the standard naming conventions for data-format-modules. If this option is omitted, the report-name (refer to -name) is used as the name of the print_order_file. If no print_order_file can be found, the user will be asked to supply a new name for the print_order_file.

-file name
-f name

The output from quick_report will be placed in the file called name. If no segment name is given, a default name of "report-name.runoff" is used. If the option is not specified, output will be printed at the user's console.

-report_header string
-rep string

where string is a sequence of ASCII characters, 128 or less in length, and enclosed in quotes if a space is used within the string. The header will be placed in the upper left hand corner of each page, along with the name of the relation. If no report_header is given the default is "QUICK REPORT".

-bks
-b

If there are backspaces and overstruck characters in the relation, the "-bks" option must be specified; otherwise, quick_report will not align the datums on their specified columns. Since checking for their presence in a relation is very time consuming, use of this option makes the program much less efficient. The default is not to check for backspaces.

-noprologue
-no

quick_report uses prologue to produce a title page for the report, number the pages, and add footers. If the user does not want these features added to the output, this option suppresses the call to prologue.

-page
-pa

When the information in the leftmost column on the page changes, a page will be ejected in the output file before continuing. For instance, if a listing of faculty by rank is desired, the data-type rank should be specified on the first line in the print-order-file. Then, with the -page option, each different rank will begin a new page. The default is not to begin a new page.

-name name
-na name

quick_report ordinarily calls prologue with the report-name "quick_report", which produces the title page and sets up conditions for runoff. With the use of this option, the user can substitute a different report-name, that is name. For example, to produce a report with the title page of a "master" report, type "-name master". The name must be listed in the relation "sms_prologue" for prologue to work correctly. The report-name is also used as the default name for the sort-order-file, print-order-file, relation-name and runoff file name.

-rt

If the system crashes during a quick_report, or the program was somehow interrupted, a relation "quickie.rel" can appear in the data base. The relation contains the same information as in the input relation, but sorted the way the sort-order-file (or print-order-file if the default is used) specified. Using this relation, the next time a report is desired saves the expense of another sort performed by quick_report. To restart, type

quick_report -rel quickie.rel -rt (other options)

-include name
-i name

where name is a runoff segment. Using the ".if" command, quick_report will insert the commands listed in the segment in the output segment, after the header lines containing the name of the relation, the report heading, the blank line, and the column header. This gives the user additional control over the format of his output. For example, if the segment contained the commands

```
.ms 2
.br
.he 3 'PAGE %'''
```

the output will be double-spaced, and header number 3 will print the page number instead of the report-header.

-line_length number
-ll number

When the "-noprologue" option is used, the four headings created by quick_report and inserted in the output segment use the runoff default line-length of 65. This may, for some purposes, be inconvenient, especially when the "-include" option is used and the line-length is changed. If this option is included, it appears as the first line in the runoff segment. All the headers will conform with the rest of the output. Both the "-line-length" and the "-include" are intended for use with the "-noprologue" option, since they replace most of prologue's function and give the user much more control over the format of his output.

Notes:

1. If the relation or print-order-file can not be found, quick report will ask

   "What is the relation?"

   and/or
   "What is the print-order-file name?"

   and wait for input.

2. quick_report looks explicitly for "quickie.rel". The "-rt" option cannot be used with any other relation as input.

3. All data-types listed in the print-order-file must appear in the sort-order-file, though the sort may be redundant.

Example:

(All responses by Multics are underlined)

1. Create sort-order-file (if necessary)

```
edm phone_book.sof
Segment not found.
Input.
name
room
extn
.
Edit.
w
q
```

2. Create print-order-file (required)

```
edm phone_book.pof
Segment not found.
Input.
/name/10/NAME
/extn/30/EXTN/extns
/room/50/ROOM
.
w
q
```

3. Execute quick_report to create runoff file

```
quick_report -name phone_book -file -report_header "EXAMPLE QUICK REPO

prologue: this report is MEMORANDUM 9999-6666

prologue: this report is to be dated August 8, 1973

prologue: this report is (e.g., PRELIMINARY, REVISED,
          FINAL) AN EXAMPLE

prologue: this report is for the (period) Summer 1973

prologue: type message, end with an additional line
          containing only a ".".
FOR DEPARTMENTAL USE ONLY
.

quickie: Finished.
```

4. runoff the file into a runout file.

```
runoff quickie.runoff -sm -in 0
```

5. Print the runout file. Normally this would be done using the Multics dprint command. For the sake of this example, phone_book.runout is shown on the following pages.

MEMORANDUM 9999-6666
August 8, 1973

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF ENGINEERING

AN EXAMPLE PHONE BOOK

Summer 1973

FOR DEPARTMENTAL USE ONLY

****

EXAMPLE QUICK REPORT
phone_book

NAME	EXIN	ROOM
Baggins, F.		
Canning, H. F.		
Gilbert, R. G.	X-3631	
Glock, R. J.		
Goldman, J. G.	X-3637	Lin--317
Lambert, P. V.		
Mason, A. H.	X-3633	Lin--303
McGary, T. X.		
Scatton, S. M.		
Sheckler, D. L.	X-3636	Lin--306
Sheckler, D. L.	X-5880	NE108-202A
Sudbury, R. W.		

(END)

Command
09/24/74

Names:

rename_set
rns
rename_set_force
rnsf

The "rename_set" command is used to change the name of a relation or data-type in the user's current data-base.

Usage:

The command may be invoked by typing:

rename_set *oldname* *newname*

or

rns *oldname* *newname*

The effect of either form results in changing the name of the relation or data-type identified by *oldname* to *newname*.

Note:

If *newname* is the name of a relation or data-type already in existence in the data-base, the command will print an error message. (The newname will not be added to the set, nor will the old set having the new name be deleted.)

Entry: rename_set_force, rnsf

This entry changes the name of a set even if a set with the new name already exists in the data base. (The name will be removed from the already existing set, even if it is the last name on the set. When the last name on a set is removed the set will be deleted.)

Usage:

rename_set_force oldname newname

or

rnsf oldname newname

The set will be renamed as for rename set, except if a set named with newname already exists in the data base it will be deleted and the new name will be given to the set, and oldname removed.

(END)

restructure_data_type

Command
09/24/74

Name: restructure_data_type, rdt

The rdt command allows the user to restructure data types managed by either astring or table data strategy modules (dsms). When restructuring an astring data type, rdt will reorganize the data elements into a balanced binary tree. For table data types, modifications such as insertions, deletions and others as described below can be made by the user to update the table.

Astring and table data types are treated differently, since, by assumption, when restructuring an astring data type only those datums which are used in some relaton are retained. For table data types, it is assumed that the user may desire a tabular data type to contain datums even though they are not presently being used in a relation. For instance, a tabular data type containing the names of the months would contain twelve (12) entries regardless of whether a particular month was being used in any relation. Moreover, since the ordering of the elements of a table data type is not computable from the datums themselves (as opposed to the alphabetical ordering assumed by astring dsms), rdt must be told explicitly where to insert new datums in an existing data type.

Usage:

rdt *data-type-name*

or

rdt *data-type-name* -options-

If the data type specified is multiply named, such as *sysr* and *name*, then each of these data types will be updated. New reference numbers will be assigned to each element of the data types, and then all relations in which these data types appear will be reconstructed so that these relations use the updated references numbers.

The options allowed with the rdt command are:

-new_dsm dsm-name

The newly created data type will be managed by the data strategy module (dsm) dsm-name, which may differ from the one associated with data-type-name. If not present, the dsm managing the old data type will be assigned as the dsm for the new data type.

-astring The dsm managing the new data type is defined to be an astring data type. If the dsm for the new data type is "dsm_astring" then this option is assumed.

-table The dsm managing the new data type is defined to be a table data type. If the dsm for the new data type is "dsm_table" then this option is assumed.

Depending upon the type of data-strategy-module (dsm), restructure_data_type must use different strategies for creating the new data-type.

Astring Restructure

Restructuring of an astring data type will cause any unused data to be deleted from the data type. All valid data elements will then be used to construct an optimal binary tree. It is expected that restructuring of astring data type will be done only in the case of a tree overflow or in periodic updating of data bases since this procedure of restructure requires a large time expenditure. Astring restructure does not require (in fact does not allow) any "requests" from the user in the manner of table restructure. Restructure_data_type will create a new data type containing only those datums used in a relation and update the relations to reflect the new reference numbers assigned to these datums.

Table Restructure

Table restructure requires the user to specify the modifications to be made to the table data type. This is done in the form of "requests" to rdt, seven of which are currently available. These requests are:

i	insert
d	delete
r	replace
m	move
c	combine
l	list
q	quit

The user may specify any of these requests following the keyword "INPUT", which will be displayed on the console when restructure_data_type is ready to accept input. After each request line, rdt will type one of the following:

1) "INPUT"
   The user may then specify another request.

2) "rdt:" followed by an error message.
   The user is informed why the previous request cannot be performed. "INPUT" will then follow.

3) "Restructure completed"

The following is a description of the "requests" available for modifying table data types. Please note that all modifications will be made on an up-to-date copy of the data type, that is, on a copy with all changes made up to this point in time. In the following description of the requests, data elements are represented by string1 or string2. If there are embedded blanks in these strings, then the data element must be enclosed in quotation marks: "string1". This is required since the parse of the input line assumes blanks as delimiters.

"i" request

Format: i string1 before string2
or
i string1 after string2

Purpose:
Insert string1 into the data type "before" (or "after", as specified) string2.

"d" request

Format: d string1

Purpose: Delete string1 from the data type.

Notes:
If string1 appears in any relation it cannot be deleted. A message will be printed.

"r" request

Format: r string1 string2

Purpose:
Replace string1 (currently in data type) with string2 (not in data type)

"m" request

Format: m string1 before string2
or
m string1 after string2

Purpose:
Move string1 immediately "before" (or "after") string2 in the data type.

Notes:
Both string1 and string2 must be members of the data type.

"c" request

Format: c string1 string2

Purpose:
Combine all references to string1 and string2 into references to string1

Notes:
string2 will be deleted from the data type.

"l" request

Format: l

Purpose:
List the data type, reflecting all updates made up to this point.

Notes:
A program_interrupt handler is established during this request.

"q" request

Format: q

Purpose:
Quit; restructure data type with all updates.

Notes:
The dsm managing the new data type may ask the user one or two questions, such as the maximum length of a datum or the maximum number of data elements allowed in the data type. When specifying the maximum length of a datum the user should allow space for any changes a data format module (dfm) might make to a datum during the generation of a report.

Example:
User input is underlined.

```
rdt acct_type
INPUT   l
          acct_type

          GEN
          FUND
          DSR

INPUT   l GRANT after FUND
INPUT   d DSR
rdt:    Datum  DSR  is  used in a relation and cannot be
deleted.
INPUT   r GEN GENERAL
```

```
INPUT 1
          acct_type

          GENERAL
          FUND
          GRANT
          DSR

INPUT 9
dsm_table: The maximum number of datums for this data
type is? 3
dsm_table: The maximum length of a datum is? 7
Restructure Completed.
```

Notes:

Restructure_data_type detects the situation in which the old and new refnos are identical. Such a situation arises when a new element is added to the end of a table data type or an element of a table data type is retyped. In this case, rdt does not have to update the reference numbers in relations and is therefore relatively inexpensive.

Rdt can not be used to create a data type or enter datums into an empty table data type. Also, astring data types which are not used in any relation can not be restructured.

Rdt will work correctly only if the relation SMS_types is correct. Rdt leaves SMS_types in a correct state. Refer to cSMSt (create_SMS_types) for information on creating an up-to-date SMS_types relation.

(END)

Names: set_data_base
sdb

set_data_base initiates a specified directory as the user's data_base for the duration of the current process, or until the next set_data_base is issued. No SMS operations may be performed until a data base is initialized.

Usage:

set_data_base pathname -control-arg-
or
sdb pathname -control-arg-

Where pathname is the absolute or relative pathname of the directory that is to be initiated as the data_base.

-control-arg- may optionally appear, and is:

-brief
-bf
this option suppresses the message informing the user that the data base specified by pathname has been successfully initiated.

Example:

set_data_base data_base.fy74

Would set the data base to a directory "data_base.fy74" in the user's working directory, and respond with a message:

data_base.fy74 has been initiated at 1542.7 est Mon 11/15/73
for User.Project

(User is the user's name and Project is the user's project.)

If instead the command line is:

sdb data_base.fy74 -brief

the data base is set but no message is printed unless an error is encountered.

Notes:

1. If no pathname is supplied and "-brief" appears, the user will be asked for a pathname of a data base.

2. If neither pathname nor -brief is specified, the current data base is terminated and no new data base is initiated.

(END)

set_file_search_rules

Command
09/24/74

Names:

set_file_search_rules
sfsr

The set_file_search_rules command allows the user to set his file search rules. File_search rules are used by the subroutine "find_file_" when looking for a segment.

Usage:

set_file_search_rules pathname

pathname — is the pathname of a segment consisting of the ASCII representation of the search rules.

Example:

```
set_file_search_rules search_rules_segment
```

or

```
sfsr search_rules_segment
```

Notes:

Each line of the segment "search_rules_segment" contains a search rule. A rule is either an absoulue or relative directory pathname, or one of the following special rules:

1) initiated_segments — This must be the first line of the search rules segment.

2) working_dir — find_file_ will search the working directory.

3) system_libraries — find_file_ will search the system libraries.

4) default_wdir — find_file_ will search the default working directory.

5) home_dir — find_file_ will search the home directory.

6) process_dir — find_file_ will search the process directory.

There must be one rule per line. A maximum of 16 search rules is allowed. Leading and trailing blanks are allowed but embedded blanks are not allowed.

The RDMS version of the "exec_com" commmand is exactly equivalent to the Multics version except it uses the file search rules to find its driving segment. (The Multics version requires an absolute or relative path specifying the driving segment.)

See also the documentation of "print_file_search_rules".

Warning: searching is expensive in machine resources so the fewer directories searched the better.

(END)

Names: sms_abbrev

This module allows use of abbreviations within the relational editor, "eds" and when using the commands mer and mart (refer to sms_interface). The set of abbreviations can be examined and redefined from command level using sms_abbrev or from within eds.

Usage:

In response to requests abbreviations can be added, deleted, or listed. While using the editor a line beginning with "¬" (a "not-sign") is interpreted as an abbrev request. From command level, a call to sms_abbrev with one argument causes that argument to be interpreted as a request:

```
sms_abbrev !
```

for example, would list the set of abbreviations, while the same effect could be obtained from within the editor using the editor request:

```
¬!
```

(Within the editor, sms_abbrev requests are distinguished by preceding them with a "not" character, ("¬").)

The available requests are:

a <abbrev> <meaning>
append the abbreviation <abbrev> having the meaning <meaning>.

d <abbrev>
delete the abbrev <abbrev>.

!

or

! <abbrev1> <abbrev2> . . . <abbrevN>

list all abbreviations, or in the second form list only the abbreviations <abbrev1>, <abbrev2>, . . ., <abbrevN>.

p

print the pathname of the segment containing the set of abbreviations. By default, this pathname is >udd>Project>User>User.sms_abbrev where "Project" is the user's project (such as EEAdmin, RDMS, Multics) and "User" is the user's login name (such as Caloggero, McGary or Wolman).

u <path>

use the segment identified by <path> as the sms_abbrev segment.

Note:

Since sms_abbrev only accepts a single argument when called from command level, requests which contain blanks must be enclosed in quotes (").

When called from command level with no arguments, sms_abbrev enters an "edit" mode. It reads requests from the console until the user types a line consisting of a period followed by a carriage return, which returns him to command level. Each line except the last must be one of the sms_abbrev requests described above.

(END)

Names: sms_error_mode, sem
print_sms_error, pse

This command allows a user to influence the action the Set Management System takes when a primitive encounters an error, and various amounts of information about the error may be recorded.

Usage:

sms_error_mode ctl_arg1 ctl_arg2 . . .
or
sem ctl_arg1 ctl_arg2 . . .

where ctl_argi may be any of the following:

-long
-lg
Record errors in long mode.

-brief
-bf
Record errors in "brief" form.

-status
-st
Print the status of error handling.

-print
-pr
Turn error printing "on".

-noprint
-nopr
Turn error printing "off".

-debug
-db
Turn calling debug (the Multics debugging program) "on".

-nodebug
-nodb
Turn calling debug "off".

By default, errors are recorded in short form, and when an error occurs it is not printed, nor is debug called.

When an SMS error occurs, the primitive in which the error occured will record the error. When an error is recorded, the following actions may be taken:

1) If error recording mode is "short", only the name of the primitive that failed will be recorded. If error recording mode is "long", in addition to the name of the primitive that failed, the argument list that caused the failure is recorded. This argument list will consist of pointers to datums (both the pointer and the datum will be printed) as well as reference numbers (of relations, data-types or datums) and options fields. The user is expected to be familiar with the calling sequences of the various SMS primitives if he/she uses the "long" mode of recording errors. In the case of the set-theoretic primitives (as discussed under the sms_interface command), the calling sequence for the SMS primitive corresponds to the sms_interface calling sequence, except that only the reference numbers for the relations involved will be printed, not their names.

2) If error printing is "on", the information just recorded will be printed on the stream "error_output". If error printing is "off", the error message is not printed at this time. (The recorded information may be printed at a later time using "print_sms_error" if it is not overwritten by another error.)

3) If calling debug is "on", a message is printed on the stream "error_output" notifying the user that the system is about to call debug, and that debug can be exited by typing ".q" followed by a newline. Then debug is invoked.

Entry: print_sms_error, pse

Usage:

print_sms_error

or

pse

The information recorded as the last SMS error is printed.

If no error has occured, a message to this effect is printed on the console. The amount of information recorded (and therefore displayed) is determined by the "long" or "brief" modes of sms_error_mode.

(END)

Names: sms_interface

The module sms_interface is a console-level, user command interface to the Relational Data Management System primitives. This document explains the meaning and use of the basic RDMS relation operations. These commands operate on two relations (files) to produce as output a result relation (file) which is a set-theoretic function of the two input relations. In the following commands, the third and fourth arguments (options and name) are optional. If a name argument is specified, then an options argument must be specified. Unless the result relation is explicitly named as the fourth argument of the user-typed command, the result relation will be a system-named relation and its reference number will be printed on the user's terminal. A system-named relation will be deleted at the end of the terminal session unless it is explicitly renamed by the user.

Usage:

user-typed command	example page
union rel-1 rel-2 options name	7
intersect rel-1 rel-2 options name	8

user-typed command	example page
difference rel-1 rel-2 options name	9
cart_prod rel-1 rel-2 options name	10
project rel-1 rel-2 options name	11
compose rel-1 rel-2 options name	12,14
sort rel-1 rel-2 options name	15
decide_over rel-1 rel-2 options name	17

user-typed command	example page
union_compose rel-1 rel-2 options name	21

where

rel-1
rel-2

are the names (or reference numbers) of the input relations or quarts, the ampersand (&), or a character string acceptable to the mqrt command discussed below. The ampersand ("&") is interpreted to mean the reference number of the relation resulting from the most recent sms_interface command.

options

is a valid options argument for the associated SMS primitive call. It is a combination of the capital letters F, S, and (Q or R), where F means "delete the first input relation", S means "delete the second input relation", and Q means "make the result a quart (temporary relation) rather than a relation". R means "make the result a relation". If neither Q nor R are supplied, the result is of the same type as the left input relation rel-1. A quart is simply a relation which is intended for temporary use for the duration of the user's terminal session (process), and is not kept permanently in the data-base, nor is it allowed to have a name. For this reason, a quart may be referred to only by its reference number. In order to facilitate this, the "&" convention is used, which means that the last result of an sms-interface command is referred to by "&", which stands for its reference number.

name

is an optional argument. If present, it will be given to the result of the SMS operation. If this argument is supplied, the options argument must be supplied. Furthermore, the result will be a permanent relation with the name as specified by the name argument.

Other Commands:

The following commands are of use in creating empty relations for sorting orders used in "sort" or datatypes used in "project" operations; small relations used for selection using the "compose" operation; or small relations used for "cart_prod" operations. In this sense, they support the use of the above relation operation commands by allowing the building of relations by simple, convenient commands without using the relation editor "eds".

mrel	name	"/data-type-1=datum-1/.../"	16

The use of "mrel", or "make_relation" will be demonstrated in the example. In brief, it creates a relation named "name" with the datatypes "data-type-1" through "data-type-k" as specified in the second argument to "mrel". If a relation of that name "name" already exists, it will be deleted and replaced by the newly created relation. If the equals ("=") following any data-type-i is used, then the command creates a one-row relation with entries "datum-1" through "datum-k" in their respective columns. Those data-type-i with no "=" (i.e. /data-type-i/) will have null datums in that column if any other datum is specified (i.e. /data-type-k=datum-k/) in order to create a one-row relation. If no equals and following datum-i are supplied, the "mrel" creates a zero-row (called null) relation.

mqrt	"/data-type-1=datum-1/.../"	16

The command "mqrt" or "make_quart" creates a quart rather than a relation. A quart is a temporary relation which is stored in a special area and cannot grow as can normal one-per-segment relations. This command creates a quart with data-types "data-type-1" etc., in the same manner as mrel. It is useful in establishing a sort-order for later use in the "sort" or "project" commands. Note that mqrt does not have a name argument, because a quart cannot have a name. This command causes

a message to be printed on the console which gives the refno of the resulting quart. For more convenience, mqrt may be used within another RDMS or sms_interface command by enclosing the mqrt command in square brackets and using it in another command line. For example, to sort the relation "alpha" by the "name" and "term" datatypes, the command line might be as follows: "sort alpha [mqrt /name/term/] R result_relation". When used as an active function, the mqrt command is first executed, creating a quart, and then the refno of the result is substituted for the entire quantity in square brackets and the command line is executed. When mqrt is used as an active function, the refno of the resulting quart is not printed, and the & default refno value is not set in order to avoid conflict with other uses of the & value.

Notes:

The reference number of the result of the most recent operation performed by sms_interface is saved internally. If either or both of the rel-1 are "&", then this saved reference number will be used as an argument.

Examples:

The following are examples describing the use and meaning of the SMS primitive relation manipulation commands. The examples are provided with their inputs, the command typed by the user, and the resulting relation. Note that in the operations compose, project, intersect, union, and difference the input relations must be sorted by the sort order of the matching datatypes of the two input relations. In this sense, the sort-order of the relation and the actual sort of the tuples of the relation are distinct concepts. A given sort-order of a relation does not always guarantee that its tuples are sorted by that ordering of datatypes, because the relation may have been modified by using the editor "eds" and not sorted when the editing was finished. During editing, unsorted and duplicate tuples may be created, and only the "sort" and "project" operations will eliminate and clean up these problems. If unsorted relations are used as inputs to these primitives, then the result relation may be unpredictable.

The set-theoretic operations of the RDMS primitive relation commands, shown below, are meaningful even for unsorted relations. However, the algorithms currently used to implement these primitives all (except for sort) are implemented to assume sorted input relations, and the output relation is guaranteed to be sorted. Thus the relation primitives are a consistent set of commands.

The command "union" forms an output relation (in this example result1) which consists of those tuples in either of the two input relations. The sort order of both relations must be the same.

directory1

sort order: name room extension

Igor ! 041 ! 3844 !
Mary ! 744 ! 3115 !
Pete ! 327 ! 3541 !
Sam ! 310 ! 3430 !

directory2

sort order: name room extension

Betty ! 110 ! 3021 !
Igor ! 041 ! 3844 !
Igor ! 744 ! 3115 !
Martha ! 242 ! 3837 !
Mary ! 744 ! 3115 !
Tom ! 243 ! 3002 !

union directory1 directory2 R result1

result1

sort order: name room extension

Betty ! 110 ! 3021 !
Igor ! 041 ! 3844 !
Igor ! 744 ! 3115 !
Martha ! 242 ! 3837 !
Mary ! 744 ! 3115 !
Pete ! 327 ! 3541 !
Sam ! 310 ! 3430 !
Tom ! 243 ! 3002 !

intersect

The command "intersect" forms an ouput relation (in this example result2) which consists of those tuple which are in both of the two input relations (directory1 and directory2). The sort order of both input relations must be the same.

directory1

sort order: name room extension

Igor	041	3844
Mary	744	3115
Pete	327	3541
Sam	310	3430

directory2

sort order: name room extension

Betty	110	3021
Igor	041	3844
Igor	744	3115
Martha	242	3837
Mary	744	3115
Tom	243	3002

```
intersect directory1 directory2 R result2
```

result2

sort order: name room extension

Igor	041	3844
Mary	744	3115

difference

The command "difference" creates an output relation (result3) which consists of those tuples that are in the first input relation (dir_of_projects) which do not match the entries in the leftmost common datatypes of the second input relation (dir_of_grades).

dir_of_projects

sort order: name room extension project

Igor	041	3844	C
Mary	744	3115	C
Pete	327	3541	A
Sam	310	3430	C

dir_of_grades

sort order: name room extension grade

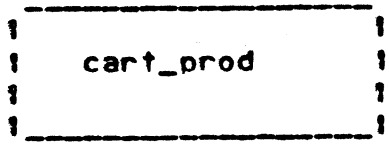
Betty	110	3021	4.5
Igor	041	3844	3.7
Igor	744	3115	4.1
Martha	242	3837	3.7
Mary	744	3115	4.8
Tom	243	3002	2.9

```
difference  dir_of_projects  dir_of_grades  R  result3
```

result3

sort order: name room extension project

Pete	327	3541	A
Sam	310	3430	C



The command "cart_prod" creates an output relation (result4) which consists of tuples generated by combining every tuple of the first input relation (directory1) with every tuple of the second input relation (term_dates). The number of tuples of the output relation (result4) is the product of the number of tuples of the first input relation and the number of tuples of the second input relation.

directory1

sort order: name room extension

Igor	041	3844
Mary	744	3115
Pete	327	3541
Sam	310	3430

term_dates

sort order: date

September 1, 1973
January 1, 1974

cart_prod directory1 term_dates R result4

result4

sort order: name room extension date

Igor	041	3844	September 1, 1973
Igor	041	3844	January 1, 1974
Mary	744	3115	September 1, 1973
Mary	744	3115	January 1, 1974
Pete	327	3541	September 1, 1973
Pete	327	3541	January 1, 1974
Sam	310	3430	September 1, 1973
Sam	310	3430	January 1, 1974

project

The command "project" creates an output relation (result5) consisting of all the distinct subtuples of the first input relation (directory2). The form of the subtuple is specified by the sort-order (data-types) of the second input relation (temporary1). In the example, the form of the subtuple is the "name" data-type, as specified by the sort-order of temporary1. Thus the output consists of all distinct "name" entries of directory2. If the second input relation does not have a sort order which matches the left part of the sort order of the first input relation the project operation does not necessarly produce either a sorted relation or a relation lacking duplicate tuples.

directory2

sort order: name room extension

Betty ! 110 ! 3021 !
Igor ! 041 ! 3844 !
Igor ! 744 ! 3115 !
Martha ! 242 ! 3837 !
Mary ! 744 ! 3115 !
Tom ! 243 ! 3002 !

temporary1

sort order: name

Igor !

project directory2 temporary1 R result5

sort order: name

Betty !
Igor !
Martha !
Mary !
Tom !

compose

The command "compose" creates an output relation in the following manner: First, it determines those left-most data-types common to both input relations (in the example the common data-type is "name"). Then, for each distinct subtuple ("name" entry) of form specified by the common data-types, which is in both input relations ("name" entries that match in project_list and directory2), every tuple of the first input relation which contains that matching subtuple is combined with every tuple of the second input relation which matches that subtuple. The meaning in a specific application is based on the user's understanding of the matching and combining with respect to the nature of the data and the meaning assigned to the user's relations.

directory2

sort order: name room extension

Betty	110	3021
Igor	041	3844
Igor	744	3115
Martha	242	3837
Mary	744	3115
Tom	243	3002

project_list

sort order: name project

Betty	A
Betty	B
Igor	C
Jean	B
Ken	B
Mary	C
Pete	A
Sam	C

```
compose directory2 project_list R result6


result6

sort order:  name room extension project

Betty ! 110 ! 3021 ! A !
Betty ! 110 ! 3021 ! B !
Igor ! 041 ! 3844 ! C !
Igor ! 744 ! 3115 ! C !
Mary ! 744 ! 3115 ! C !
```

This example of the compose command illustrates its use as a means of selection or retrieval. The second input relation (which is created by the sms_interface program, and automatically deleted after it is used) contains one "name" entry, and is used to select that part of directory2 that matches that "name" entry (Igor). The output of the compose, select1, contains those selected tuples from directory2.

directory2

sort order: name room extension

Betty	110	3021
Igor	041	3844
Igor	744	3115
Martha	242	3837
Mary	744	3115
Tom	243	3002

```
compose directory2 /name=Igor/ R select1
```

select1

sort order: name room extension

Igor	041	3844
Igor	744	3115

The "sort" command sorts the first input relation by the sort-order of the second input relation. Note that the data-types of the second input relation must be included in those of the relation being sorted. Also note that the data of the second input relation is not used, and, like the "project" command, merely specifies a sort-order for the operation.

directory1

sort order: name room extension

Igor	041	3844
Mary	744	3115
Pete	327	3541
Pete	329	3025
Sam	310	3430

sort directory1 /name/extension/room/ R result7

result7

sort order: name extension room

Igor	3844	041
Mary	3115	744
Pete	3025	329
Pete	3541	327
Sam	3430	310

mrel, mqrt

The command "mrel" stands for "make relation", and is used to create small relations for the purpose of selection using compose, etc., without having to use the editor. Note that the first argument to mrel is the name of the relation to be created. If a relation of that name already exists, it will be deleted and replaced by the result of this mrel command. If a quart is desired, the command "mqrt" may be used instead. The usage is identical to "mrel", but no relation name is supplied - the reference number of the result is printed on the console. By enclosing the command mqrt in brackets "[...]", it may be used as an active function and used as part of other RDMS command lines. When used as an active function, [mqrt ...] does not print the refno on the console, and the default refno value & is not set to the result of the mqrt.

```
mrel result8 "/room=041/name=Igor/"


result8

sort order:  room name

041 ! Igor !



mqrt "/name/project/extension/"

make_quart: quart with refno 382 created.

382

sort order:  name project extension

no rows
```

decide_over

The command "decide_over" provides a one-pass numerical decision and simple assignment facility for RDMS. It may be viewed as a means of subsetting or selecting from a relation those rows (tuples) which meet any of a set of condition "terms" specified in the "condition relation". A "term" is a set of conditions which are applied to a row of the input relation, and those conditions must all be true in order for the term to be true, and for that row to be copied into the output relation of the decide_over operation. A "term" example might be the following: (name = "Doe, John" and percent = "50%"). Both of these conditions must be satisfied by the row of the input relation in order for that row to be copied into the output relation. The "condition relation" is a set of terms, and therefore might consist of the following terms: (name = "Doe, John" and percent = "50%"), (name = "Smith, Bill" and percent = "40%"). Given an input relation with "name" and "percent" columns, then the output relation will consist of all rows of the input for which either the first or the second term is true. Details of the usage of decide_over and more specifics of its operation are described below.

Usage:

```
decide_over  input_rel  condition_rel  options  output_rel
```

The first relation argument "input_rel" is the relation over which the conditional subsetting operation is to be performed. The second relation argument "condition_rel" specifies the conditions and the data-types or columns of the input relation to which those conditions apply. Each row of the condition relation specifies a "term" which is applied to the currently tested row of the input relation. The condition relation has the columns as shown below:

c1 <dt1> c2 <dt2> c3 <dt3> ... ck <dtk>

where the datatype names <dt1>, <dt2>, etc. specify the columns of the input relation which are tested, and the entry in the c1 column is the condition with respect to the entry in the <dt1> column of the condition relation, as applied to the entry in the <dt1> column of the input relation. The example above would have a condition relation of the following form:

c1	name	c2	percent
=	"Doe, John"	=	"50%"
=	"Smith, Bill"	=	"40%"

In this example, the condition-argument pair (=,"Doe,John") is applied to the "name" datatype of the input relation. To be more explicit, each row r of the condition relation is a term which says that, given row m of the input relation, if each (i=1,...,k) condition-argument pair (ci,<dti>) of row (term) r is "true" as applied to row m of the input relation, then row m is copied into the result relation in the form:

<dt1> <dt2> <dt3> ... <dtk>

In the example above, the output relation will have two columns: name and percent.

Several other requirements and features should be mentioned. The condition datatypes c1, c2, c3, etc. must all be names on a datatype managed by dsm_char4. The condition pairs of each row (term) r of the condition relation are all of the following form:

<condition> <argument>

where <condition> is one of the following set of valid conditions: (*,=,~,>,<,G,L) and where <argument> is any element of the datatype <dti>. The input relation is examined in a single pass one row at a time, and every condition term is then applied to test that row. The result relation is built up by appending to it each input row, for each term that it satisfies.

If a given input row yields a "true" when applied to to more than one term of the condition relation then that input row will be copied more than once. This feature is useful because decide_over also provides the ability to make simple assignments which are specific to the particular term which is "true". This will be explained more fully later.

The meanings of the condition specifiers are as follows:

*	don't care (column entry of input row always passes)
=	equals (column entry must equal condition argument)
-	not equal (column entry must not equal condition argument)
>	greater than (column entry must be greater than condition argument)
<	less than (column entry must be less than condition argument)
G	greater or equal (column entry must be greater than or equal to condition argument)
L	less or equal (column entry must be less than or equal to condition argument)

Another example of the use of decide_over is the following. This will explain the use of the * (don't care) condition. Consider the following condition relation:

c1	name	c2	semester
*	""	=	"FIRST TERM"

Using an input relation with column datatypes "name" and "semester" will simply copy all rows of that input relation for which the entry in the "semester" column of that row is "FIRST

TERM", regardless of the entry in the "name" column of that row. This allows the "name" column to be preserved in the output relation without applying any conditions on "name" entries.

As another example, consider the condition relation of two terms as shown below:

c1	name	c2	role	c3	percent
*	""	=	recitation	<	50%
*	""	~	tutorial	>	10%

This condition relation consists of two terms, and each row of the input relation is "copied" for each of these terms that it satisfies. In this example, each row of the input relation is copied (in just the name, role, and semester columns) if (role = recitation and percent < 50) or if (role ~= tutorial and percent > 10%). If both terms are true or satisfied, then that row of the input relation will be copied twice. In this case, a row of the input relation will be copied twice if role = recitation and 10% < percent < 50%. Note that the (*, "") condition-argument pair is used simply to copy the "name" part of the input row; no tests are applied to the name entries of the input relation.

union_compose

The union_compose operation operates much like the compose operation, but does not lose information when there is no match between tuples of either relation. Tuples which match in the common columns of the two input relations are handled identically to compose. A tuple of the first input relation which does not match any tuples of the second relation is copied into the output relation as is, but with null datums copied into those columns not in the first input relation. The same is done with the tuples of the second input relation which match no tuples of the first. Refer to the compose example for the relations project_list and directory2. Note the differences between the output of the union_compose and compose operations.

```
union_compose directory2 project_list R result11


result11


sort order:  name room extension project

Betty ! 110 ! 3021 ! A !
Betty ! 110 ! 3021 ! B !
Igor ! 041 ! 3844 ! C !
Igor ! 744 ! 3115 ! C !
Jean !  !  ! B !
Ken !  !  ! B !
Martha ! 242 ! 3837 !  !
Mary ! 744 ! 3115 ! C !
Pete !  !  ! A !
Sam !  !  ! C !
Tom ! 243 ! 3002 !  !
```

(END)

Name: sms_star

This command/active-function extends the Multics star convention for use in the RDMS environment. All data base objects whose names match a given star-name can be listed, or a more restricted group of objects, (such as all relations or all data types or both) can be matched against the star name.

Usage:

This program may be invoked either as a command or as an active function. If called as a command, the results are printed on the console on separate lines. When invoked as an active function, the result is the qualifying names concatenated together separated by blanks.

Use as a command:

sms_star <star_name> <control_arguments>

Use as an ative function:

[sms_star <star_name> <control_arguments>]

1) <star_name>

is optional. If it appears, it is a star-laden name which will be matched against names of RDMS relations, data types, rsms, dsms and system objects. By default a <star_name> of "**" is used. This <star_name> matches any name on any object in the data base. The types of objects the star name will be matched against can be restricted; see the discussion of <control_arguments>.

2) <control_arguments>

are also optional. They may be taken from the "selector" control arguments and the "other" control arguments described below.

A) Selector <control_arguments>

By default all types of objects are matched against the star name. If one or more of the following selector <control_arguments> appear, however, only the objects of the types mentioned are matched against the star name.

-set

If this control argument appears, relations and data-types are matched against the star_name. Using this control argument is equivalent to using all of "-relation", "-data_type" and "-quart".

-relation
-rel

The names of all relations will be matched against the star_name.

-data_type
-dt

The names of all data types in the data base are matched against the star_name.

-quart
-qt

The names of all quarts are matched against the star name. (Note: quarts are currently unnamed, so the character string representation of their reference numbers is used as their "name" for sms_star. Note also that quarts are not in the data base, but are temporary objects that can be used only by one user, and which are deleted when the user logs out.)

-strategy_module
-sm

Strategy modules (dsms and rsms) are matched against the star_name. This is equivalent to including "-dsm" and "-rsm".

-dsm

Data strategy modules are matched against the star name.

-rsm
Relational strategy modules are matched against the star_name.

-system
-sys
System relations and data types are matched against the star_name. This is equivalent to using both "-sdt" and "-srel".

-sdt
System data types are matched against the star name.

-srel
System relations are matched against the star name.

B) Other <control_arguments>

In addiion to the <control_arguments> for selection described above, some special <control_arguments> also exist:

-nnl
This control argument appiies only when sms_star is used as a command. Normally the qualifying names are printed on separate lines. This arguments causes the names to be printed as one long line, separated by blanks.

-primary_name
For objects having more than one name, only the first of the names which match the star name is used. (Normally all names which match the star name are used.)

Examples:

1) To discover the names of all relatons for which there exist "editing copies", the command line:

```
sms_star eds.** -rel
```

will list all relation names starting with "eds." (This is how editing copies of a relation are named. For example, the editing copy of the relation x would be named "eds.x".)

2) To list all objects in the data base,

```
sms_star
```

(invocation with no arguments) will suffice.

3) The active function form can be very useful in obtaining a list of names to be supplied to some other command. Suppose a user wanted to know the sort order of all the relations in the data base currently set. The command line:

```
status_sets ([sms_star -rel]) -sort
```

would invoke the "status_sets" command once for each relation in the data base.

(END)

Name: sms_type

This command/active-function determines the type of an SMS object, given its reference number or name.

Usage:

sms_type xxx
or
[sms_type xxx]

where xxx is either the name or reference number of an object in the presently set data base. The types of objects possible are:

relation
data-type
quart
rsm
dsm
system-relation
system-data-type
system-special
invalid

If the type returned is "invalid", the name or reference number xxx does not identify an object in the data base.

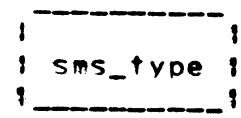
If no data base has been set, an error message is printed, and if sms_type has been invoked as a command the condition "command_error" is signalled. If invoked as an active-function, the condition "active_function_error" is raised.

Example:

Suppose the currently set data base contains a data type named "room", having a reference number of 82. Then the following examples might occur. Lines typed by the user are identified by "Un)", responses by the system by "Sn".

U1) sms_type room
S1) sms_type: room is of type 'data-type'

U2) sms_type 82



S2) sms_type: 82 is of type "data-type"

U3) loa_ [sms_type room]
S3) data-type

U4) loa_ [sms_type 82]
S4) data-type

U5) sms_type xxxxxxx
S5) xxxxxxx is of type "invalid"

Notes:

See the introductory section of this manual for a description of the various types of objects which may exist in a data base.

(END)

Name: status_sets, sts

This command provides information on the size, sort order, date and time last modified and dumped, and lastly the strategy module of a data type or rleation.

Usage:

status_sets **name** opt-1. . .opt-n

or

sts **name** opt-1. . . opt-n

1) **name**
is the name or refno of a set or module.

2) opt-i may be taken from the following:

-size
Print the size of the set: for relations, this is the number of rows and the number of columns: for data types this is the number of data elements in the data type.

-sort
If the set is a relation, print the current sort order. For data types, this control argument has no effect.

-date
Print the date and time information for the segment containing the set: when it was created, when it was last modified, and when it was dumped.

-sm
Print the strategy module of the relation or data type.

-status <status-argument>

<status-argument> will be passed to the Multics status command to obtain information about the segment containing the set. A leading "-" will be added to <status-argument> when it is passed to status, and thus <status-argument> should not begin with a "-". See the documentation for the Multics "status" command.

-all

Returns -size, -sort, -sm, and -date for the given relation or data type, as well as all information about the segment containing the set that can be obtained using the "status" command.

Notes

By default, status_sets prints the name, length, order (and for relations the sort order) when no control arguments are supplied.

This command accepts the star convention.

Examples

1) The command line:

```
sts phone_book
```

would result in:

```
        phone_book
length = 637     order = 3

name  room  extn
```

(END)

terminate_data_base

Command
09/24/74

Name: terminate_data_base, tmdb

This command cleans up a data base and then terminates the data base. That is, it deletes all quarts and temporary relations, and then leaves the user with no data base set.

Usage:

terminate_data_base

(there are no arguments)

Example:

terminate_data_base

Notes:

See the documentation of "cleanup_data_base". The cleanup_data_base command cleans up a data base but leaves the data base initiated.

(END)

Names: terminate_set, tmset

This command causes the Set Managment System to terminate the data type, relation or strategy module named. This means that at the next reference to the set or module, SMS will search for the object named and remember its location. All succeeding references will use the location remembered, and will not search for the object by name. Once a "set" has been referenced, all succeeding references would use the pointer set by SMS at the first reference, unless the name is terminated using terminate_name.

Usage:

terminate_set set
or
tms set

where set is the name of the set or strategy module to be terminated.

Example:

```
tmr dsm_char4
terminate_set dsm_char4
initiate my_dsm_char4 dsm_char4
list_data_type char4_data_type
```

These commands might be used by a programmer testing a new version of the data strategy module "dsm_char4". The call to list_data_type to list the contents of char4_data_type (which is assumed to be managed by dsm_char4) requires that dsm_char4 be found if it has not been found already. The use of terminate_set followed by initiate ensures that the desired program will be used.

Notes:

Before using this command, a user should understand the idea of initiation and termination in the Multics environment.

(END)

translate_column

Command/Active Function
09/24/74

Name: translate_column, tc

translate_column is used when it is desired to reassign the reference numbers assigned to datums in a particular data-type in a single relation. The main purpose of this command is to aid in the use of ciphered information. The ciphered information is assigned reference numbers which do not preserve the sorting order of the information; thus, to sort relations with the ciphered information different reference numbers must be assigned which do preserve the sorting order. Usually this deciphered relation will only be a temporary relation and the default action by translate_column is to create a quart with the reassigned reference numbers.

Usage:

translate_column rel_name -options-

where:

rel_name must be the first argument to translate_column. This is the name of the relation which is copied and has a column's (or more than one column's) reference numbers reassigned.

-options- is taken from the following list:

-data-type dt-name new-dt-name
-dt dt-name new-dt-name

At lease one occurrence of this option is required. dt-name specifies which column (data-type) of the relation rel_name is to have its reference numbers reassigned. This argument specifies the name of the data-type to use in reassigning reference numbers. If this arguemnt is omitted then translate_column will attempt to use the data-type name "dt-name.tc". The -dt argument may occur more than once, so that multiple columns can be translated with one invocation of translate_column.

-relation output-rel
-rel output-rel

This argument specifies that the result of translate_column should be put into a relation (instead of a quart). The output-rel argument is optional. If present it is the name to place on the resulting relation; otherwise, the relation will be named +TEMP+.unique (where unique stands for 15 unique characters). If output-rel is equivalent to rel_name then the result of translate_column will replace the original relation.

-brief
-bf This argument suppresses the message giving the reference number of the resulting relation. When invoked as a active function, translate_column does not print this message.

Example:

```
translate_column sal_list -dt salary

translate_column: relation with reference number 383 created.
```

The above example reassigned reference numbers to the datums in the salary data-type column of the sal_list relation. The datums were assigned reference numbers in the data-type "salary.tc". The resulting relation (which is a quart) has a reference number of 383.

```
loa_ [tc sal_list -dt salary new_salary]
384
```

In the above example, translate_column is used as an active function to again reassign reference numbers to the salary column of relation sal_list. This time, however, the user explicitly specified which data-type to use (new_salary). The result of the active function is the character string representation of the reference number of the result of translate_column.

Note:

When translate_column is invoked as a command (as opposed to an active function), the default reference number maintained by sms_interface is set to the result of translate_column. Refer to the writeup of sms_interface for more information on the default reference number.

(END)

Name: tree_stuff

This command creates an optimum binary tree in a data-type segment managed by dsm_astring, given an alphabetically sorted list of ASCII strings in a file. After an astring data type has been created, tree_stuff is used to "prime" the data type for most efficient insertion of new information and to ensure that the tree will not be likely to overflow. (An astring tree is said to overflow when while attempting to add a datum to the data type, there is no reference number available for the new datum even though all possible reference numbers have not been used. If (to use an oversimplified case) "Allen" had been assigned the reference number 6, and "Asimov" the reference number 7, and it is desired to insert the datum "Arthur", we find that although Arthur should follow Allen and precede Asimov, there is no reference number available which is greater than Allen's reference number while less than Arthur's. Even though it may be possible to insert "Mullen" and obtain a reference number, we say the tree has overflowed because "Arthur" could not be added.)

Usage:

tree_stuff *data-type* *file-name*

Notes:

1) *data-type* must have been created prior to usage of tree_stuff and must be managed by dsm_v2_astring.

2) A data base containing *data-type* must be initialized prior to usage of tree_stuff.

3) The ASCII text segment *file-name* must be in the current working directory as data_type (*file-name* is not a data type or relation). It consists of up to 1024 lines of text, each of which represents one data element to be inserted in

data-type. The file may be created by either edm or qexd (or a program) and should be alphabetized before usage of tree_stuff. The sort_file command (see MPM) will accomplish the alphabetic sort.

Example:

```
tree_stuff  name  namelist
```

This takes the data elements from the ASCII file "namelist" and place them in an optimum binary tree in the data type "name".

(END)

Name: where
wh

This command/active-function searches for object segments using the currently defined search rules. If the segment is found, the pathname is either printed on the console (command) or returned as the value of the active function.

Usage:

where name1 . . . nameN

or

wh name1 . . . nameN

When used as a command, where prints the pathnames of the object segments on separte lines. If used as an active function, the pathnames are concatenated, separated by blanks, to form the return string.

Example:

1) where eds

Response:
>udd>RDMS>service>bound_editing_

2) loa_ [string [where eds where]]

Response:
>udd>RDMS>service>bound_editing >udd>RDMS>service>where

Notes:

If the segment identified by name1 is not found, an error message is printed on the console. If the use is as an active function, the condition "active_function_error" is then raised.

(END)

Names:

whod

This command lists users of the Relational Data Management System who currently have data bases set.

Usage:

whod -user-list- -ctl-args-

-user-list-
is optional. If it appears it contains one or more terms of the form: "User.Project", "User" or ".Project". (A term "Goldman.RDMS" will cause the user Goldman to be listed if he is logged in on the project RDMS and has a data base set; a term "Goldman" will cause Goldman to be listed if he is logged in under any project and has a data base set; finally, a term ".RDMS" will cause any users logged in on the RDMS project to be listed if they have a data base set. Note that the -user-list- may include several terms, such as ".MITASD .RDMS .EEGradOff " if it is desired to list which of these projects is using a data base. If no -user-list- appears, all users who have data bases set will be listed.

-ctl-args-
is also optional. If it appears it may contain any or all of the following:

-l

lists users and their data bases in long form. This includes the date and time the user logged in, his terminal id, whether the user is absentee (indicated by "(A)"), secondary("(S)") or normal (indicated by blank), and finally the user's name, the user's project and the user's data base.

-a

list only absentee users.

Example:

```
whod -a
```

might produce the following output on the console:

```
Caloggero.EEAdmin (A)     >udd>EEAdmin>Caloggero>db.fy74
```

Note:

See the related commands "whop", "hmd", and "hmp".

(END)

Names: whop

This command lists users of the Relational Data Management System who are currently logged in, along with the data base a user currently has set (if any).

Usage:

whop -user-list- -ctl-args-

-user-list-
The -user-list- is optional. If it appears only the users specified by it will be listed The -user-list- consists of one or more terms of the form "McGary.RDMS" (list McGary if he is logged in on the RDMS project), "McGary" (list McGary if he is logged in under any project), or ".RDMS" (list any user of the RDMS project who is logged in). If no -user-list- appears, all persons logged in who are using the Relational Data Management System will be listed.

-ctl-args- may be none or any number of the following:

-long Normally the users are listed in "short" form, i.e., the user's name, the project he is logged in under and the data base he is using (if any) will be listed. In long form the information includes:
1) the date and time the user logged in (e. g., "11/12/73 1548.0")
2) the user's console I.D. ("543", "a327", or "none")
3) the type of user (absentee denoted by "(A)", secondary by "(S)", normal user by blanks,
4) the user's name ("Dodge" or "Goldman")
5) the user's project ("EEGradOff" or "RDMS")
6) the data base the user has set (">udd>EEGradOff>Dodge>db.eeg" or if no data base is set, the field is blank).

-d Only users who have data bases set will be listed. The default is to list all users.

-a Only absentee users will be listed.

-p <path> Instead of the standard "whotable" (which records who is using the Relational Data Management System) the "whotable" specified by <path> will be used in checking for users. This is primarily a debugging aid.

Example:

```
whop
```

might cause the following to be printed on the console:

```
Caloggero.EEAdmin            >udd>EEAdmin>Caloggero>db.fy74
McGary.RDMS
Goldman.RDMS                 >udd>o>dbs>tdb.fy72
TJohnson.DFIS
```

while

```
whop -lg
```

might produce:

```
11/10/73 0905.0 543  Caloggero EEAdmin  >udd>e>rjc>db.fy74
         1015.5 none McGary    RDMS
         1047.9  none Goldman   RDMS     >udd>o>dbs>tdb.fy72
         1107.3 a213 TJohnson  DFIS
```

Notes:

See the related commands "whod", "whmp", and "hmd".

(END)