A700

RIM Users Manual

by Jim Fox

University Computing Services
University of Washington

Revision date: February 9, 1990

This document tells you how to use Rim—University Computing Services' relation database management system, available on all of the mainframe computers at the UCS. You are assumed to be able to access the computer of your choice, and be able to edit text files. Chapter 6, which discusses the program interface, assumes you are familiar with fortran-77.

RIM Users Manual
UCS UCS Document A700
Revision Date February 9, 1990

Contents

List of Figures

No Warranty

Because Rim is licensed free of charge, we provide absolutely no warranty, to the extent permitted by applicable state law. The University of Washington, University Computing Services, or other parties provide Rim "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should Rim prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event unless required by applicable law will the University of Washington, University Computing Services, or any other party who may modify and redistribute Rim as permitted above, be liable to you for damages, including any lost profits, lost monies, or other special, incidental or consequential damages arising out of the use or inability to use (including but not limited to loss of data or data being rendered inaccurate or losses sustained by third parties) the program, even if you have been advised of the possibility of such damages, or for any claim by any other party.

Chapter 1

Introduction

Rim is a popular and easy-to-use relational database management package available on all of the University Computing Services mainframe computer systems. It was specifically designed to be easily transported between computers and operating systems without the need for customization. Rim users will find the same command language, functionality, and reliability in all implementations.

A relational database may be thought of as a collection of one or more tables. These tables (also called *relations* in database terminology) consist of rows (*tuples*) and columns (*attributes*). This book will use the terms table, row,

and column.[1] Rim allows the values in a table to be scaler, vector, and matrix integer or real numbers; character strings; dates; or times. The vectors, matrices, and character strings may be of fixed or variable length.

A table is defined with a fixed number and sequence of columns. The **conductors** table in figure 1.1, for example, is defined with four columns: **Atomic_No.**, an integer; **symbol**, a fixed length character string; **resistivity**, a double precision real number; and **name**, a variable length character string. As data are added to, deleted from, or changed in the table the number and sequence of rows will vary, but the number and sequence of the columns will remain the same. The **conductors** table contains five rows.

atomic_no.	symbol	resistivity	name
29	Cu	12.2	Copper
13	Al	12.9	Aluminum
26	Fe	20.3	Iron
–NA–	LB	–NA–	Bernstein
92	U	–MV–	Uranium

conductors table

Figure 1.1: The `conductors` table of the Sample database. "-NA-" indicates that the column is not applicable for the row. "-MV-" indicates that a value for the column is missing. The complete Sample database is shown in figure 3.3 on page 24.

The primary advantage of the relational database is its structural simplicity. There is no need for the user to learn the parent-child relationships found in hierarchical databases, nor is there need to understand the pointers of a network database. Data are always represented by simple tables.

[1] The term 'column' is also used for the columns of a matrix. This may cause hesitation, but should not cause confusion.

The Structured Query Language (SQL), pronounced '*see-quel*', is a command syntax for relational database access. It uses keywords and an "English" syntax to construct database commands. For example,

```
select symbol, resistivity from conductors where
    resistivity lt 14;
```

is a SQL command which selects those rows from the "conductors" table that have a resistivity less than 14. SQL has been proposed by the American National Standards Institute (ANSI) as the basis for a standard relational database language.

Rim does not attempt to implement a precise or complete SQL. Instead, it uses SQL as a guide, so its command language may be best described as "SQL-like". Although you can tailor Rim to accept the particular SQL command above,[2] the analogous, normal Rim command is

```
select symbol  resistivity from conductors where   +
    resistivity lt 14
```

It has few of minor differences from the SQL syntax: a plus sign indicates line continuation and no explicit command terminator is required. Chapter 3 describes the Rim command language in detail.

The principal advantage to Rim users of a language like SQL (versus screens or menus) is the universality and commonality of the dialog interface. Rim's appearance is the same, regardless of the computer, terminal, or operating system. A second advantage is that users already familiar with SQL will not have to learn a new language to use Rim.

[2]The 'comment' `*(set continue=null)` accomplishes this feat. See section 3.13.3 for details on the 'comment' commands.

An algebra is defined for relational databases which will create new tables from old. Here is a brief look at the operations of this algebra:

union: The resultant table contains all rows from two source tables.

intersection: The resultant table contains only those rows from two source tables where like columns have equal values.

subtraction: The resultant table contains only those rows from one source table which are not also in the other source table.

join: The resultant table's rows contain columns from two source tables. The rows are matched by comparing a specified column in each table.

projection: The resultant table is a subset of a single source table. It may have fewer columns and fewer rows than the original table.

These operations are discussed in detail in section 3.11, which describes the Rim relational algebra commands.

Rim provides a convenient function interface for users who need to access a Rim database from within their programs. The interface provides one Fortran-77 function the program to issue Rim commands and another function transfers data between Rim and the user's program. Chapter 6 describes the Rim program interface in detail.

Rim provides a convenient program interface, relational data access, and much support for loading, unloading, maintaining, and reporting on databases. It allows your application to be very portable—as long as you keep it clean of system dependencies. It does not, however, present a very convenient user interface for adding or changing data. And it has no screens.

Rim is most useful for databases that are not updated frequently, OR for dynamic databases that will be updated by user programs.

Rim is a descendent of the Boeing Computer Services program of the same name that was developed in 1978 as part of the IPAD project (NASA contract NAS1-14700). That program was brought to the University of Washington and further developed as UWRIM, a CDC Cyber program. The present program has been rewritten in completely portable fortran-77 code. The new code includes new functionality and a more "SQL-like" command language. Some capabilities of UWRIM are no longer supported.

Chapter 2

What is a Rim database?

This chapter describes a Rim database in detail.

The Rim database consists of all the column and table definitions, all of the table data (rows), and supplementary information such as passwords.

Many elements of a Rim database are identified by a name: the database name, owner and table passwords, table names, and column names. These names consist of one to sixteen characters (letters, digits, and the underscore are allowed). They may contain both upper and lower case letters but Rim will always disregard the case of letters when comparing the names. '**conductors**' and '**Conductors**' are valid names. They are also identical. '**my text**' is not a valid name because a space is not allowed.

A column in a Rim database identifies both a particular column of a table and the type of data contained in that column. The column **symbol** in the Sample database (figure 3.3 on page 24), for example, identifies particular columns in the "conductors", "measures", and "notes" tables. It also represents a data type of text with a length of eight characters.

These are the data types supported by Rim.

text A text column is a fixed or variable length character string. Rim uses the ASCII character representation on all machines.

integer An integer column is a one-word integer.

integer vector An integer vector column is a fixed or variable length, one-dimensional integer array.

integer matrix An integer matrix column is a two-dimensional integer array. It may have either fixed length rows and columns, fixed length rows and variable length columns, or variable length rows and columns.

real A real column is single floating point number.

real vector A real vector column is a fixed or variable length, one-dimensional array of floating point numbers.

real matrix A real matrix column is a two-dimensional array of floating point numbers. It may have either fixed length rows and columns, fixed length rows and variable length columns, or variable length rows and columns.

double A double column is single, double-precision, floating point number.

double vector A double vector column is a fixed or variable length, one-dimensional array of double-precision, floating point numbers.

double matrix A double matrix column is a two-dimensional, double-precision array. It may have either fixed length rows and columns, fixed length rows and variable length columns, or variable length rows and columns.

date A date column is a Julian integer value. It is input and reported in a user specified format (including year, month, and day).

time A time column is a integer value containing the number of seconds from midnight. It is input and reported in a user specified format (including hour, minute, and optional second).

Each column has an optional default format, which will be used by Rim during formatted input or output if no explicit format is otherwise specified.

A Rim table is defined as a sequence of columns. Rows may be added to or deleted from the table, but the order and number of columns is invariant.

Each row of the table contains one data item for each column.[1] The data may be an actual value or it may be a 'missing value' code. The **conductors** table in Figure 3.3 is an example of a table with four columns and five rows.

[1] Note that one data item for a vector or matrix column may consist of several numbers, and one data item for a text column may consist of many characters.

A link is a logical connection between each row of a source table and a unique row of a destination table. It is most commonly used to associate an identifier, **tech_id** in the **measurements** table of figure 3.3, for example, with information about that identifier—a row in the **technicians** table in this case.

Access to tables is optionally protected by passwords.[2] Rim provides three levels of password protection:

owner password, specified by the **define owner** sub-command, restricts access to database modification commands and to protected tables. Users who have not entered the owner password (**user** command) are unable to modify the database and are required to enter the appropriate table password for access to protected tables. Users who have correctly entered the owner password (**user** command) have unrestricted access to the database.

table read password, specified by the **define passwords** sub-command, prevents unauthorized users from reading the protected table. Permission to read is granted only to those users who have entered either the owner password or the table's read password.

[2]Users may also be restricted by permissions granted, or not granted, by the operating system.

table modify password, also specified by the **define passwords** sub-command, prevents unauthorized users from modifying the protected table. Permission to modify is granted only to those users who have entered either the owner password or the table's modify password.

See section 3.3 and 3.13.2 for complete descriptions of the **owner** and **user** commands.

A key is an ordered list of pointers to the rows of a table—much like the index of a book. Existence of a key for an column can greatly facilitate Rim's access to data rows during retrievals. However, because keys need to be maintained, their existence is detrimental to Rim's efficiency during database update.

Normally you will build a key for those columns of a table which are most often referenced in query commands. Rim will, in some circumstances, automatically build a key. Columns for which keys have been built are referred to as "keyed columns".

Occasionally a partial row must be added to a table. This partial row will have data for some columns but will not have data for others. Rim provides two distinct missing values, distinguished by their codes: '**-MV-**', for 'missing value', and '**-NA-**' for "not applicable" value. The **conductors** table of the Sample database (page 24) contains one missing value and two values that are not applicable.

The database is contained on three direct access files on the user's disk area. The actual naming of these files is dependent upon the capabilities and conventions of the particular operating system. Generally the files have a common name, which is the database identifier, and extensions of `rimdb1`, `rimdb2`, and `rimdb3`. The content of these files is the responsibility of Rim, but the user is responsible for any copies or backups of the database. The files are not in text format and cannot be directly edited.

Chapter 3

Using Rim

This chapter describes the command language of Rim. As discussed in the introduction, this language is similar to SQL but is not an exact implementation. To see how to run Rim on your computer, consult Appendix A.

A Rim command consists of a command name that is sometimes followed by keywords and qualifiers (table names, column names, data values, filenames, etc.). Input is free format with one or more spaces delimiting the items. A trailing plus sign (+) continues a command on the next line.

In addition to the space and plus, these characters have special meaning to Rim.

`( ) [ ] , ; : < > @ % =` and sometimes ' "

You must enclose these characters in quotes (either '—' or "—") to enter them as text.

This book uses the following conventions to describe the syntax of Rim commands.

Bold text

indicates that you must type the command or keyword exactly as shown. Except that you may abbreviate keywords to three characters. For example, **sel** is a suitable abbreviation for **select**.

Italic text

indicates that you must enter the name of something For example, if *table* is specified, you must supply the name of a specific table.

⟨*Angle brackets*⟩

denote an optional word or phrase.

Stacked
items

indicate that you must enter one of the items in the stack.

...

indicate that you may enter more than one of the previous item.

⋮

indicate that you may enter more than one of the previous line.

In addition, `typewriter text` is used to indicate specific examples of commands.

Many of these conventions may be used in a single command description. This example

select $\begin{matrix} column\langle[col\langle,row\rangle]\rangle\langle@fmt\rangle \ \ldots \\ * \end{matrix}$ **from** *table*

indicates that these are valid commands.

```
select name from conductor
select index@i2 name from conductor
select * from conductor
```

Of course the [*col*] and [*col*,*row*] are valid only for vector and matrix columns, respectively.

Rim supplies online help text which describes the syntax and functionality of commands, typing conventions, and also provides general information about Rim. Enter

help

to view an introduction to Rim and a list of Rim commands. Enter

help typing

to see help text about Rim's command and data entry conventions. Enter

help ⟨*command* ⟨*sub-command*⟩⟩

to see help text related to a specific Rim command.

You may include a comment nearly anywhere in your Rim input by enclosing it in '*(' and ')'.

part of command *(*This is a comment*) rest of command

You may not put comments in quoted text strings or in formatted input data. The Rim commands which load the Sample database (figure 3.6 on page 29) contain comments.

There is also a comment command.

* *rest of command*

The *rest of command* is parsed, so normal input rules must be followed, but no action is taken. This command is used for comments, but is also useful with the **echo** mode setting (page 45) to show macro expansions (see chapter 4).

This section tells you how to create, change, link, and remove tables. You must have entered the owner password (if the database has one) before you can use database definition commands.

Create new tables, or change the definition of existing tables, with a block of commands that begins with "define" and ends with "end". The block is show in figure 3.1.

define *filename* ⟨file parameters⟩
⟨**name** *name*⟩
⟨**owner** *password*⟩
⟨ **columns**
column definitions ⟩
⟨ **tables**
table definitions ⟩
⟨ **links**
link definitions ⟩
⟨ **passwords**
password definitions ⟩
end

Figure 3.1: The block of commands that begin with **define** and end with **end** define or modify the definition of a database.

define ⟨*filename* ⟨*parameters*⟩⟩

begins definition of a new database if the files do not exist, or begins modification of an existing database if the files do exist. Some systems allow a parameter following the filename to further identify the files.[1] Consult Appendix A to see if your system allows this parameter.

If you are defining a new database, the files are created and the database name becomes the filename.

[1] For instance, VM/CMS users can specify the files' filemode after the name.

If a database is currently open, the filename may be omitted and the current database will be edited.

Enter the define command

name *name*

to give the database a new name. This does not affect the files on which the database resides.

Enter the define command

owner *password*

to identify the owner password of the database. Use this command only if

1. your user password is not the same as the owner password of an existing database, or
2. you want to add an owner password to a new or existing database.

Enter the define command

columns

Data type	Syntax	Definitions
text	**A***xx*	*xx* = field width for formatted input and output. Output text longer than *xx* characters will be continued on subsequent lines. Input text longer than *xx* characters will be truncated.
int *real*	⟨*r*⟩**I***n*⟨**.***d*⟩ ⟨*r*⟩**F***n*⟨**.***d*⟩	*r* = repeat count for vector and matrix output. Output longer than *r* numbers will be paragraphed. Input longer than *r* numbers will be truncated. *n* = field width per number. *d* = decimal places. If *D* is specified for integer input, the value will be multiplied by 10^d after input. If *D* is specified for integer output, the value will be divided by 10^d before output.
date	*string*	*string* = a reasonable combination of **dd**, **mm** or **mmm**, and **yy** or **yyyy**. **mm/dd/yy** and **mmm-dd-yyyy** are valid date formats.
time	*string*	*string* = a reasonable combination of **ss** (optional), **mm**, and **hh**. **hh:mm:ss** and **hh:mm** are valid time formats.

Figure 3.2: Column format syntax

to begin entry of column definitions. All entries until the next **define** subcommand are column definitions. There are four styles of column definition lines. They have the general syntax

name type ⟨*length*⟩ ⟨**format** *format*⟩

where *format* is the optional, default format for the column. Table 3.2 shows the syntax of format specifications.

Define a text column with

name **text** $\begin{matrix} \#chars \\ \textbf{var} \end{matrix}$ ⟨**format** *format*⟩

where **var** indicates a variable length string. Table 3.2 shows the syntax of format specifications.

Define a scaler column with

$$\textit{name} \begin{matrix} \textbf{int} \\ \textbf{real} \\ \textbf{double} \\ \textbf{date} \\ \textbf{time} \end{matrix} \langle \textbf{format } \textit{format} \rangle$$

Define a vector column with

$$\textit{name} \begin{matrix} \textbf{ivec} \\ \textbf{rvec} \\ \textbf{dvec} \end{matrix} \begin{matrix} \#cols \\ \textbf{var} \end{matrix} \langle \textbf{format } \textit{format} \rangle$$

where **var** indicates a variable length vector.

Define a matrix column with

$$\textit{name} \begin{matrix} \textbf{imat} & \#rows, \#cols \\ \textbf{rmat} & \#rows, \textbf{var} \\ \textbf{dmat} & \textbf{var, var} \end{matrix} \langle \textbf{format } \textit{format} \rangle$$

where **var** indicates a variable length matrix row or column.

Enter the define command

tables

to begin entry of table definitions. All entries until the next **define** subcommand are table definitions. Each entry contains a table name followed by a list of columns

name **with** *column* ⟨*column* ⟨...⟩⟩

where each *column* has been defined in the **columns** section or already exists in the database.

Enter the define command

links

to begin entry of link definitions. All entries until the next **define** subcommand are link definitions. Each has the following syntax.

link name **from** *att1* **in** *rel1* **to** *att2* **in** *rel2*

where the value of column *att1* in table *rel1* will be used to find a unique row of table *rel2*, where *att2* = *att1*.

Links are used by the **select** command.

Enter the define command

end ⟨**define**⟩

to end database definition and returns to normal Rim command processing with the newly defined database open.

Figure 3.4 on page 25 shows the input that defined the tables in the sample database.

You can change the name of a table or column with the **rename** command. To rename a table enter

rename table *table* **to** *new_name*

To rename a link enter

rename link *link* **to** *new_name*

To rename a column enter

rename ⟨**column**⟩ *column* **to** *new_name* ⟨**in** *table*⟩

If a table name is given, the column is renamed only in that table. Otherwise the column is renamed in all tables.

You can change the default format of a column with the **reformat** command.

reformat *column* **to** *format* ⟨**in** *table*⟩

changes the default format for the specified column. If a table name if given, the format is changed only in that table. Otherwise it is changed in all tables.

You can remove a table or link from the database with the **remove** command.

remove ⟨**table**⟩ *table_name*

removes the specified table. All data rows contained in the table will be lost. Any links to or from the table will also be removed. If you are running Rim interactively, Rim will ask for confirmation before removing a table.

Removes a link with the command

remove link *link_name*

Sample

Figure 3.3 on page 24 shows an example database that is used with this reference manual. Figure 3.4 shows the input that was used to create the Sample database. Figure 3.5 shows the input that was used to modify the Sample database.

atomic_no.	symbol	resistivity	name
29	Cu	12.2	Copper
13	Al	12.9	Aluminum
26	Fe	20.3	Iron
–NA–	LB	–NA–	Bernstein
92	U	–MV–	Uranium

conductors table

id	name	position
5	John Jones	Tech 1
22	Jim Smith	Tech 2
35	Joe Jackson	Tech 1

Techs table

symbol	id	date	time	resistivity
Cu	5	88-01-21	08:10	11.9
Cu	5	88-01-21	10:32	12.4
Al	35	88-02-10	13:45	13.0
Al	35	88-02-11	09:34	14.3
Cu	22	88-02-22	08:48	12.5
Fe	5	88-03-04	15:33	19.4

measures table

symbol	id	date	time	notes
Cu	5	88-01-21	09:03	Spilled coffee on sample. Expect higher resistivity due to poor electrical contact.
Cu	22	88-02-22	09:20	Sample seems contaminated with some sort of dusty deposit.

notes table

Figure 3.3: The Sample database used by all the example commands in this reference manual.

```
define Sample
columns
  at_no         int             format i4
  symbol        text     8
  resistivity   double          format f7.6
  name          text     var    format a12
  M_date        date            format 'yy-mm-dd'
  M_time        time            format 'hh:mm'
  id            int             format i5
  position      text     8
  status        text     1
tables
  techs with id name position status
  conductors with symbol at_no resistivity name
  measures with symbol id M_date M_time resistivity
links
  M_C from symbol in measures to symbol in conductors
  M_T from id in measures to id in techs
end
```

Figure 3.4: Initial definition of Sample database

```
define Sample
columns
  notes         text     var    format a25
tables
  notes with symbol id M_date M_time notes
links
  notes_c from symbol in notes to symbol in conductors
end
```

Figure 3.5: Modification of Sample database definition

open

You access an existing database by opening it with this command

open *name*

Some systems allow you to specify file pathnames or directory names to locate the database files. Consult Appendix A for details.

When Rim opens a database it also looks for an input file with the same name as the database and with an extension of **.rim**. If this file exists it is assumed to contain Rim commands and will be input automatically. This initialization file is commonly used to load database specific macros (See chapter 4).

close

Rim will close an open database as it exits, but you may also manually close it at any time with the close command

close

You can load data rows into tables from two types of sources.

Free-format is the form of most hand-typed data. In free-format sources:

- data items are delimited by spaces and commas;
- data on the input records are in the same order as the columns of the table; and
- each input record must exactly fill a table row (The input record may, of course, span several lines if each is continued with a +).

Fixed-format is the form of most computer generated data. In fixed-format sources:

- data items are located in fixed columns—no delimiters are required;
- numeric data may appear anywhere in a field—leading and trailing blanks are ignored;
- leading blanks in text data are always retained;
- trailing blanks in text data are stripped and do not affect the length of variable length text columns;
- data on the input records need not be in the same order as the columns of the table;
- each row may span several records (The + is not applicable);
- some columns of the input records may be ignored; and
- some rows of the table may not be filled (These will become "missing values").

In order to load data, you must have entered either the owner password or the modify password (MPW) for the table you are loading.

This form will load data either from the terminal or from a file. The command block

load *table*
data records
end ⟨**load**⟩

loads the data records into the specified table.

If the data records are in a file, use the command

load *table* **from** *filename*

to load the table. In this case the trailing **end** is optional. Rim will assume the end if it reaches the end-of-file. See the notes on filenames in appendix A. Figure 3.6 shows the input that loaded most of the Sample database.

This form is almost always used with the data in a file. Fixed-format loading requires a format definition block that describes the position and format of each column to be loaded. It is also usually in a file and looks like

format
line position column_name format
⋮
end ⟨**format**⟩

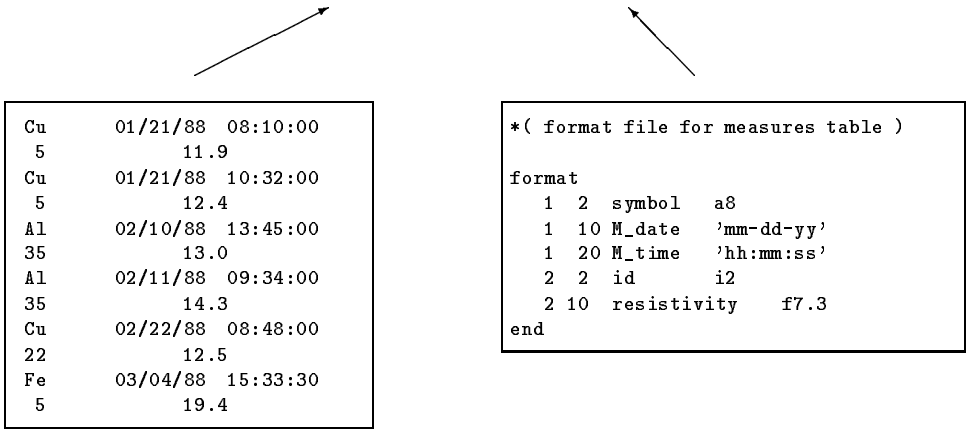
where *line* is 1 for the first input record of a row, 2 for the next input record, etc. *Position* is the starting character position in the record for the data. Data formats are described in figure 3.2. You may enter a null field (' ') in place of the *column_name* and *format* to cause Rim to skip an input line.

```
*( Load the Sample database )
open Sample
load conductors
  Cu 29 12.2 Copper
  Al 13 12.9 Aluminum
  Fe 26 20.3 Iron
  LB -NA- -NA- Bernstein
  U 92  -MV- Uranium
end

load techs
  5  'John Jones'   'Tech 1'  A
  22 'Jim Smith'    'Tech 2'  A
  35 'Joe Jackson'  'Tech 1'  A
end

*(The colon is a Rim delimiter and cannot be
  used in unquoted strings)
*(Since Rim ignores non-essential fields in dates and times,
  a dot is used as the spacer in this fields.)

load notes
  Cu  5  88/01/21  09.03  +
  'Spilled coffee on sample.  Expect higher readings due +
to poor electrical contact.'

  Cu 22  88/02/22  09.20  +
    'Sample seems contaminated with bitter deposit. +
Recommend addition of cream and sugar.'
end
```

Figure 3.6: Free-form loading of Sample database.

Note that the input data is in normal Rim form—comments are allowed, fields are separated by spaces or commas, strings are enclosed in quotes, and lines are continued with a trailing plus sign.

```
load measures from mxxxx.dat using measures.fmt
```



```
Cu      01/21/88  08:10:00
 5            11.9
Cu      01/21/88  10:32:00
 5            12.4
Al      02/10/88  13:45:00
35            13.0
Al      02/11/88  09:34:00
35            14.3
Cu      02/22/88  08:48:00
22            12.5
Fe      03/04/88  15:33:30
 5            19.4
```

```
*( format file for measures table )

format
   1  2  symbol   a8
   1  10 M_date   'mm-dd-yy'
   1  20 M_time   'hh:mm:ss'
   2  2  id       i2
   2 10  resistivity    f7.3
end
```

Figure 3.7: Fixed-form loading of Sample database.

Note that the format file is normal Rim form but the data file is not.

Begin formatted loading with

load *table* **from** *data_file* **using** *format_file*

where the data is in file *data_file* and the format block is in file *format_file*. The latter may be "**terminal**" if you want to enter the format block online. If both filenames are the same Rim will first read the format and then the data from the same file. See the notes on filenames in appendix A.

Figure 3.7 shows an the input that loaded the **measures** table.

select

This section shows you how to retrieve data stored in your database. The **select** command, which has many clauses and optional forms, is the easiest

way to look at your data. If you need more flexibility, or more complicated reports, you should use the report writer (Chapter 5). The **select** command has the general format

select *column specifications* +
⟨**from** *relation*⟩ +
⟨*where clause*⟩ +
⟨*sort clause*⟩ +
⟨**to** *filename*⟩

where the last three clauses are optional. If you omit the **from** clause Rim will use the most recently accessed table. If there is no such table Rim will report an error.

This is a list of the columns to display, along with an optional description of how to display them. The column may be either from the main table (as indicated on the **from** clause) or may be from a linked table. In its simplest form a column specification is just a list of columns.

```
select id m_date from measures
```

```
id     M_date
-----  --------
    5  88-01-21
    5  88-01-21
   35  88-02-10
   35  88-02-11
   22  88-02-22
    5  88-03-04
```

If the column to be displayed is in a linked table, specify the link name first, followed by a colon and then the column name.

link_name:*column_name*

where *column_name* refers to a column in the linked table.

```
select symbol id M_t:name M_date from measures
```

symbol	id	name	M_date
Cu	5	John Jones	88-01-21
Cu	5	John Jones	88-01-21
Al	35	Joe Jackson	88-02-10
Al	35	Joe Jackson	88-02-11
Cu	22	Jim Smith	88-02-22
Fe	5	John Jones	88-03-04

You may specify a format for each column if you don't want to use Rim's default. After the column name type an "at" sign (@) and the format.

column_name@*format*

The format syntax is described in figure 3.2.

If you do not specify a format Rim will use a default for the column—as specified either by **define** commands or by the **reformat** command. If neither of those was specified Rim will use a default format according to the column type. You may display these defaults with the **show** command (page 44) and may set them with the **set** command (page 45).

```
select symbol id@i4 M_t:name +
m_date@'dd-mmm-yy' from measures
```

```
symbol    id    name          M-date
--------  ----  ------------  ---------
Cu           5  John Jones    21-JAN-88
Cu           5  John Jones    21-JAN-88
Al          35  Joe Jackson   10-FEB-88
Al          35  Joe Jackson   10-FEB-88
Cu          22  Jim Smith     22-FEB-88
Fe           5  John Jones    04-MAR-88
```

You may specify the title for each column if you don't want to use Rim's default. After the column name type a percent sign and the title.

column_name%title

Title should be enclosed in quotes.

```
select symbol id%'Tech id' M_t:name +
    m_date@'dd-mmm-yy'%'Measure date' from measures
```

```
symbol    Tech id  name          Measure date
--------  -------  ------------  ------------
Cu              5  John Jones    21-JAN-88
Cu              5  John Jones    21-JAN-88
Al             35  Joe Jackson   10-FEB-88
Al             35  Joe Jackson   11-FEB-88
Cu             22  Jim Smith     22-FEB-88
Fe              5  John Jones    04-MAR-88
```

You may total a column by following it with an equal sign and an 'S'.

column_name=**S**

sums the column. There is no useful summation in the Sample database to use for an example.

You may select all columns of a table, with the default formats and titles, with an asterisk (*) in place of the column specifications.

```
select * from measures
```

symbol	id	M_date	M_time	resistivity
Cu	5	88-01-21	08:10	11.900
Cu	5	88-01-21	10:32	12.400
Al	35	88-02-10	13:45	13.000
Al	35	88-02-11	09:34	14.300
Cu	22	88-02-22	08:48	12.500
Fe	5	88-03-04	15:33	19.400

Rather than selecting individual rows from a table, you can have Rim perform simple tallies on columns. The general form of the function **select**

is

select ⟨*column*⟩ *function*(*column* ⟨...⟩) ⟨...⟩ **from** *table*

where the functions may be

num() sum() ave() max() or **min()**

each with an obvious functionality. Only one detail column may be specified. If it is omitted the selection will produce a single tally for the table. Functions will always completely disregard any missing values.

```
select symbol m_c:name num(resistivity) +
    ave(resistivity@f9.2) from measures
```

symbol	name	NUM(resistivity)	AVE(resistivity)
Al	Aluminum	2	13.65
Cu	Copper	3	12.27
Fe	Iron	1	19.40

The **where** clause acts as a filter which passes only those rows that pass the clause criteria. For example,

select *columns* **from measures where symbol='Cu'**

selects only the measurements of copper (Cu) and ignores the other rows. The **where** clause consists of

comparisons of the form

column op value
column op column
column test

where *op* and *test* are described in figure 3.8,

boolean operators which combine comparisons

comparison **and** *comparison*
comparison **or** *comparison*
not *comparison*

parentheses which specify the precedence of the boolean operations

$(C_1\ op_a\ C_2)op_b\ C_3$ $C_1\ op_a\ C_2$, then $op_b\ C_3$
$C_1\ op_a\ (C_2\ op_b\ C_3)$ $C_2\ op_b\ C_3$, then $C_1\ op_a$

```
select symbol m_c:name resistivity@f7.2 from measures +
    where symbol='Cu'
```

```
symbol   name          resistivity
-------  ------------  ------------
Cu       Copper          11.90
Cu       Copper          12.40
Cu       Copper          12.50
```

The **sort** clause specifies the sort order of the displayed rows. It's form is

sort ⟨**by**⟩ *column* $\left\langle \begin{array}{c} =\mathbf{A} \\ =\mathbf{D} \end{array} \right\rangle$ ⟨...⟩

op	alt	$V = A$ *op* B
eq	=	V is true if $A = B$
ne	<>	V is true if $A \neq B$.
ge	>=	V is true if $A \geq B$.
gt	>	V is true if $A > B$.
le	<=	V is true if $A \leq B$.
lt	<	V is true if $A < B$.
like		V is true if $A \in B$. (See note 1)

test	$V = A$ *test*
exists	V is true if A is defined.
fails	V is true if A is undefined. A missing value or a not applicable value is undefined. All other values are defined.

1) The **like** comparison is for strings only. The strings are compared character-by-character, however the B string for this comparison may contain 'wild-card' characters. A '**?**' in B will match any single character in A, while an '*****' in B will match any number of characters in A. The 'wild-card' characters may be changed with the SET ARBCHAR command. This comparison may be used only with a value—*column* **like** *column* is not valid.

2) String matching is also sensitive to the setting of the **case** setting, which determines whether or not upper and lower case letters match one another. The default is **case respect** which says that upper case letters do not match lower case letters.

Figure 3.8: **where** clause comparisons

where **=A** indicates an ascending sort order (the default) and **=D** indicates a descending sort order. The first column in the clause is the primary sort column; the second is the secondary sort column; etc.

A **sort** clause may not be specified with a function **select**—the independent column is necessarily the sort column.

```
select symbol m_c:name resistivity from measures +
    sort by symbol
```

```
symbol    name          resistivity
--------  ------------  ------------
Al        Aluminum       13.000
Al        Aluminum       14.300
Cu        Copper         11.900
Cu        Copper         12.400
Cu        Copper         12.500
Fe        Iron           19.400
```

Keys

This section shows you how to speed up your data retrievals by building keys for some commonly accessed columns. A key is a list of pointers to the data records of a table. The key is ordered by the values of the column for which the key is defined. Whenever Rim is performing an "equals" type of search, it will use the pointers to access data records, resulting in a much faster retrieval. However, updates to the table are more time consuming when columns are keyed. When possible, you should first load the table and then build any useful keys.

Build a key for a column with this command.

build key for *column* **in** *table*

Usually, columns that are targets of links will be keyed columns. The Sample database should have these columns keyed:

```
build key for symbol in conductors
build key for id in techs
```

In addition, if the **notes** table becomes large, it might be reasonable to build keys for both **symbol** and **id** in **notes**.

Remove a key for a column with this command.

remove key for *column* **in** *table*

This section shows you how to change the data values in your tables. Use the change command:

change *column* **to** *new_value* **in** *table* **where** *where_clause*

which changes the value of *column* to *new_value* for every row of *table* that satisfies the **where** clause.

This section shows you how to delete rows from your tables. Use the delete command:

delete rows from *table* **where** *where_clause*

which deletes all rows in *table* that satisfy the **where** clause.

If you want to transfer your database to another computer (which supports Rim), or if you want to save it in a system-independent form you can **unload** it to a text file. You may unload all or selected tables, and either the definitions, the data, both, or the passwords. Use the command

$$\textbf{unload} \left\langle \begin{array}{c} \textbf{all} \\ \textbf{definitions} \\ \textbf{data} \\ \textbf{passwords} \end{array} \right\rangle \langle \textit{table} \ ... \rangle \ \langle \textbf{to} \ \textit{filename} \rangle$$

to unload your database. The default is to unload both definitions and data (all) and to unload all tables. The unloaded file is in normal text format with lines no longer than 80 column per line.

You can reload the unloaded database simply by inputting the text file into Rim.

Notice that this command also gives you a means to reload your database. You might want to do this when it accumulates a large number of deleted records.

To reload your database:

1. Unload the database to a text file.
2. Backup and purge the database files.
3. Input the text file to Rim to rebuild the database.

You can use relational algebra on tables in your database to create new tables. This section describes Rim's relational algebra commands.

The **union** command creates a new table which contains all or selected columns from two source tables. The resultant table contains all rows from each source table. When rows from the two tables match in all common columns, a single resultant row is created. When rows from the two tables do not match in all common columns, a resultant row is created for each source row, with missing values filling out the rows.

union *table-1* **with** *table-2* **forming** *table-3* +
⟨**using** *column-1 column-2 ...*⟩

creates *table-3* from the union of *table-1* and *table-2*.

The **intersection** command creates a new table which contains all or selected columns from two source tables. The resultant table contains only those rows from each source table which match in all common columns.

intersect *table-1* **with** *table-2* **forming** *table-3* +
⟨**using** *column-1 column-2 ...*⟩

creates *table-3* from the intersection of *table-1* and *table-2*.

The **subtract** command creates a new table which contains all or selected columns from two source tables. The resultant table contains only those rows from the second table which do not match all common columns of the first table.

subtract *table-1* **from** *table-2* **forming** *table-3* +
⟨**using** *column-1 column-2 ...*⟩

creates *table-3* from the subtraction of *table-1* from *table-2*.

The **join** command creates a new table which contains all columns from two source tables. The resultant table contains rows from each source table where specified columns match according to the specified comparison.

join *table-1* **using** *column-1* **with** *table-2* **using** *column-2* +
forming *table-3* **where** *comparison*

creates *table-3* from the join of *table-1* and *table-2*. Each combination of rows where

column-1 in *table-1* *comparison* *column-2* in *table-2*

is true, creates a row in *table-3*.

The **project** command creates a new table which contains all or selected columns from a source table. The resultant table contains all or selected rows from the source table.

project *table-1* **from** *table-2* ⟨**using** *column-1 column-2 ...*⟩ +
where *conditions*

creates *table-1* from *table-2*. *table-1* contains selected columns (**using**) and the selected rows (**where**) of *table-2*.

Rim normally read commands from your terminal and writes back to your terminal. You can tell Rim to read from a file by entering the command

⟨**set**⟩ **input** *filename*

where *filename* identifies the file to read. It is specified in the standard format for your system. On systems which commonly use two-part filenames

an extension will be assumed if it is not specified. Consult Appendix A to see if Rim will assume an extension for your input files. An 'end-of-file' generates an automatic **end** command.

You can redirect Rim's terminal output to a file by entering the command

⟨**set**⟩ **output** *filename*

where *filename* identifies the file to write to. On systems which commonly use two-part filenames an extension will be assumed if it is not specified. Consult Appendix A to see if Rim will assume an extension for your output files.

Rim is governed by many parameters. Many of these represent fixed limits. You can see what these limits are with this command

show limits

You have direct control over many other parameters. You can see these with this command

show

You can also use the **show** command to view macro definitions (see section 4.4 on page 53).

The parameters you may set are:

set name *new name*

sets the database name to *new name*. This is a permanent change to the database.

⟨**Set**⟩ **user** *password*

sets the user password.

⟨**Set**⟩ **echo** ⟨ **on** / **off** ⟩

sets the command echo mode. When echoing is on, all commands are 'echoed' to the terminal, or to the report file if output has been redirected.

Set case **ignore** / **respect**

sets the case mode for text matching. If case is **ignore**, lower and upper case letters will match. If case is **respect**, lower and upper case letters will not match.

Set single arbchar *character*

sets the 'single' wild-card character for **like** string matching. Occurrences of *character* in the template string will match any single character in the target string. The default value for this character is a question mark (**?**).

Set multiple arbchar *character*

sets the 'multiple' wild-card character for **like** string matching. Occurrences of *character* in the template string will match any number of characters

(including none) in the target string. The default value for this character is an asterisk (*).

Set MV *string*

sets the **MV** missing value string for data input and output. **MV** missing values will appear in reports as *string*. Occurrences of *string* in input data will load the corresponding column with a type **MV** missing value. If the *string* is blank, any blank input field will be assigned the MV missing value. The default value for MV is '`-MV-`'.

Set NA *string*

sets the **NA** missing value string for data input and output. **NA** missing values will appear in reports as *string*. Occurrences of *string* in input data will load the corresponding column with a type **NA** missing value. If the *string* is blank, any blank input field will be assigned the NA missing value. The default value for NA is '`-NA-`'.

Set { **integer** | **real** | **date** | **time** } **format** *format*

sets the default format for the selected item.

Set terminal width *number*

sets the width of the terminal to *number* characters. Lines output to the terminal will not be longer than this value.

Set report width *number*

sets the maximum width of reports to *number* characters.

Set report height *number*

sets the page height for reports to *number* lines. If *number* is zero, there will be no pagination—no headers, no footers, and no forms control.

You can maintain a log of your commands by **tracing** them to a file.

Set trace ⟨**to** *file*⟩ / **off**

turns tracing on or off.[2]

You may also change some of the rules Rim uses to parse your input. Since you must be able to enter special characters, Rim requires you to put these commands inside 'comments'. The parsing options are specified as

***(set** *option*=*character*)

where the *character* is any character or **null**, and *option* is one of the following.

del sets the alternate field delimiter (other than space). If it is set to null, space is the only delimiter. The default is a comma (**,**).

con sets the line continuation character. When this character appears at the end of an input line the command is continued on the next line. If it is set to null, all lines are continued and a command ends only at the **end** character. The default is a plus (+).

end sets the end of command character. Where this character appears in the input line the command is completed. The next command may follow on the same line. If it is set to null, all commands must end at the end of a line. The default is a semicolon (**;**).

[2]Tracing is actually a more robust feature than is described here. Is is used to diagnose Rim problems. See the *Rim Installers Manual* for more detailed information.

For example,

```
*(set del=/) *(set con=null)
```

indicates that the virgule (/) will separate fields and that all commands will end only at semicolons.

You may want to recall, edit, and reuse a previous command, either to fix an error or to add or delete parameters. Or you may want to use an old command as a basis for construction of a new command. Entry of a single

r

on the command line will recall the most recently entered command *line* for edit and execution.

When the recalled command is displayed, you may

- enter another '**r**' to recall the next most recent command,
- enter return to execute the command, or
- edit the command. Characters typed replace corresponding characters in the command, except that

  space retains a character,
  # deletes a letter,
  < begins insertion of characters,
  > ends insertion of characters, and
  ! truncates the line.

  The edited command will be displayed and you may make further edits or execute it.

Chapter 4

Defining and Using Macros

A macro is a word that has the meaning of several words. More specifically, a macro has a name and a definition. When Rim encounters a macro name during text input, it replaces the name with the macro's definition.

Define a macro with the command

macro *name* = *definition*

where

name	=	name of the macro
definition	=	replacement text

Here is an example which illustrates macro usage.

Define the macro **person** with the command

```
macro person = 'id@i4 m_t:name'
```

Then you can use the name **person** in selection commands, such as this example.[1]

```
select symbol person +
    m_date@'dd-mmm-yy' from measures
```

symbol	id	name	M-date
Cu	5	John Jones	21-JAN-88
Cu	5	John Jones	21-JAN-88
Al	35	Joe Jackson	10-FEB-88
Al	35	Joe Jackson	10-FEB-88
Cu	22	Jim Smith	22-FEB-88
Fe	5	John Jones	04-MAR-88

Rim has interpreted **person** to mean `id@i4 m_to_t:name`.

The **person** macro in the previous example provides a convenience, but has limited utility. The macro's meaning never changes. Often you want a

[1] Compare this to the command on page 32.

slightly different meaning each time the macro is invoked.

To gain this flexibility define a macro with arguments. The following example demonstrates such a macro.

Define the macro **rename** by the command

```
macro rename = 'change name to "' 2 '" in techs ' +
    'where id = ' 1
```

With this definition you can use the new 'command' **rename**[2] to change names in the **techs** table. For example,

```
rename 5 'John A. Jones'
```

which Rim will interpret as

```
change name to "John A. Jones" in techs +
    where id = 5
```

The macro definition text consists of text fields and integer argument numbers (range 1–31). When the macro is invoked, by the occurrence of its name in input text, the text portions of the macro's definition are copied verbatim. The integers in the definition are replaced by corresponding argument fields

[2]Even with the above definition of the macro **rename**, you can still use Rim's **rename** command to rename tables, links, or columns. You just have to abbreviate the command. Rim will only recognize a macro when its name is typed in full.

following the macro's name in the input. No spaces are added between fields in the definition text. For example, with the definition

```
macro alfa = 'abc ' 2 ' def' 1 '; xxx'
```

the input

```
select alfa this that from ...
```

is interpreted by Rim as

```
select abc that defthis; xxx from ...
```

Note that there were no spaces between '**def**' and the argument number (**1**). If you want separation spaces around the arguments, as with argument **2** here, you must put them in.

Note also that this macro (**alfa**) contained an end-of-line character (**;**). Macros may expand into several Rim commands.

If the **alfa** macro had been invoked with fewer arguments, for example,

```
select alfa this
```

it would be interpreted by Rim as

```
select abc defthis; xxx from ...
```

The unused arguments are simply ignored.

A more sophisticated macro example is discussed in the report writing chapter, on page 75.

Macros are expanded prior to being **echo**d—assuming **echo** has been set. You can look at the expansion of any macro by **echo**ing a comment command (*). For example, assuming the **rename** definition on page 51, you could enter

set echo
*** rename x Anyone**

and Rim would echo

```
change name to Anyone in techs where id = x
```

You can also look at any or all of your macro definitions with this **show** command

show macro ⟨*name*⟩

If *name* is given, only that macro will be displayed.

You clear (delete) a macro definition with the command

macro *name* **clear**

Clearing macros recovers macro definition space.

Chapter 5

Printing Reports

Rim's flexible report writer lets you produce sophisticated reports on your database. You have complete control over page headers, footers, and detail lines. You can combine data from several tables in a single report. And you can alter the format of the report based on the actual content of individual rows in the database.

Before learning to use the report writer you should be familiar with the Rim commands described in the previous chapter. In particular, you must thoroughly understand the **select** command.

A Rim report is the product of several commands, which are first 'compiled' by Rim and then 'executed' as a block to produce the report. You therefore

rarely actually type report writing commands directly into Rim. Instead, you should enter them into a separate text file—using the local system editor—and then **input** them to produce the report.

Here is an example input file and the report it produces to illustrate Rim's report writing statements.

```
report
  header
    1 1 'symbol'
    1 10 'date'
    1 20 'resistivity'
    2 1 ' '
  end header
  select from measures sort by symbol
    print
      1 1 symbol a8
      1 10 m_date 'yy/mm/dd'
      1 20 resistivity f8.2
      2 1 ' '
    end print
  end select
end report
```

```
symbol   date         resistivity

Al       88/02/10     13.00
Al       88/02/10     14.30
Cu       88/01/21     11.90
Cu       88/01/21     12.40
Cu       88/02/22     12.50
Fe       88/03/04     19.40
```

Notice first that the report definition consists of blocks, which begin with a '*command*' and end with '**end** *command*'. The indentation is intended to accentuate this block structure and you are advised to follow this practice in your own reports.

In this example, the **select** block identifies the rows and sort order of the rows of the *measures* table that are to be extracted (this example selects all rows). The **print** block describes the actual format of data on the report. The **report** block contains the entire report definition.

Report writer commands are called as *statements* to distinguish them from other Rim commands.

The report writer allows you to define variables, assign values to them, and use them much like columns of a table. The assignment statement

$$\textit{variable} \left\langle \begin{array}{c} \textbf{int} \\ \textbf{real} \\ \textbf{double} \\ \textbf{date} \\ \textbf{time} \\ \# \end{array} \right\rangle = \textit{expression}$$

defines *variable* (if it hasn't yet been defined), gives it a type (**int** is the default, # is the length of a text variable), and assigns it the value of *expression*. Variable names are similar to column names.

The *expression* is any rational (to Rim) combination of values, table columns, and variables connected by the mathematical operators +, -, *, and /, with parenthesis allowed to specify operator precedence. This is much like the assignment statement of any common programming language. For example,

```
temp_id = id
xsym 8 = symbol
next_day = m_date + 1
ave_error = (tot_resist - ave_resist) / count
```

are valid assignment statements. These rules govern expressions.

1. Text data may be concatenated with the + operator.
2. Vector and matrix elements may be specified by *name*(*item*) or *name*(*row column*) in standard Rim fashion.
3. If both operands are integers, the arithmetic and the resultant will be integer.
4. If either operand is real or double, the arithmetic and the resultant will be double precision.
5. The default operator precedence is left to right in all cases. For example,

   a+b*c ≡ **(a+b)*c**.
6. When a real value is converted to an integer, the value is truncated.
7. String values are either padded with blanks or truncated to match the length of the receiving variable.
8. Date and time columns are treated as integers.
9. Any occurrence of "missing values" in an expression gives the expression a "missing value".
10. There are four pre-defined variables:

    page_number – the number of the current page,
    line_number – the number of the current line,
    report_date – the date at the start of report execution, and
    report_time – the time at the start of report execution.

report

The report block defines the entire report.

report
⋮
end ⟨**report**⟩

When Rim sees the **report** statement, it begins compiling the report. When it sees the **end report** statement, it begins to process the stored commands. Any error during the compilation phase will terminate compilation and the report will not be printed.

select

The selection block identifies a table, selection criteria for selecting rows from that table, and a sort order for row selection. The **select** statement is very similar to Rim's **select** command except that it contains only the **from**, **where**, and **sort** clauses.

select from *table* ⟨**where** *criteria*⟩ ⟨**sort** ⟨**by**⟩ *sort order*⟩
⋮
end ⟨**select**⟩

where

table may be any table in your currently open database,

where ... is a normal, valid Rim **where** clause (see section 3.6.4) that may also contain variable names in addition to values and column names from *table*, and

sort ... is a normal, valid Rim **sort** clause (see section 3.6.5).

All statements within the selection block, which may include other **select** statements, will be processed for each row of *table* that passes the **where** criteria. Rows are retrieved and processed in the order specified by the **sort** clause. For example, the statements

```
yesterday = report_date - 1
select from measures where M_date = yesterday
   .
   .
   .
end select
```

selects all measurements from the day before the report date.

print

Data are actually printed on the report by print blocks. Print blocks specify the data to be printed, their formats, and their positions within the block.

print
line position $\begin{matrix} column \\ variable \end{matrix}$ *format*
line position text_string
⋮
end ⟨**print**⟩

Each print block produces a specific number of lines—which is at least the maximum of the *line* values, but may be greater if an item is paragraphed.

line is the relative line number within the block on which this item is to begin printing. The first line is 1.

position is the horizontal position on the line to begin printing this item. The first position is 1.

format is a valid format (see section 3.6.1).

Paragraphed items will continue on following lines. Make be sure you don't print anything directly beneath a paragraphed item.

This report program

```
report
  select from notes sort by id
    print
      1 5 id     i2
      2 5 symbol a8
      1 15 m_date 'yy/mm/dd'
      2 15 m_time 'hh:mm'
      1 25 notes   a20
    end print
    print       *(print a blank line)
      1 1 ' '
    end print
  end select
end report
```

produces this report

```
 5        88/01/21  Spilled coffee on
Cu        09:03     sample.  Expect
                    higher readings due
                    to poor electrical
                    contact.

22        88/02/22  Sample seems
Cu        09:20     contaminated with
                    bitter deposit.
                    Recommend addition
                    of cream and sugar.
```

One of the things a selection block may contain is another selection block. The inner block is processed completely for *each* selected row of the outer block. Column names always refer only to the 'current' table—column names from other tables are inaccessible. All defined variables are always available.

In programming terminology we say that column names are 'locally' defined and variables are 'globally' defined.

This report program

```
report
  select from techs where position = 'Tech 1'
    print
      1 1 id     i4
      1 10 name  a20
    end print
    mid = id  *(save id for measures selection statement)
    select from measures where id = mid sort by symbol
      print
        2 5 symbol a8
        2 15 m_date 'yy/mm/dd'
        2 25 resistivity f6.2
      end print
    end select
    print
      1 1 ' '
    end print
  end select
end report
```

produces the following report

```
   5     John Jones

    Cu        88/01/21   11.90
    Cu        88/01/21   12.40
    Fe        88/03/04   19.40

  35     Joe Jackson

    Al        88/02/10   13.00
    Al        88/02/10   14.30
```

A header is text printed at the top of each page of a report. A footer is text printed at the bottom of each page of a report. They are printed if, and only if, the report height parameter is non-zero and the header (or footer) block has been 'executed'. Following are definitions of the **header** and **footer** blocks.

header
line position text_string
line position variable format
⋮
end ⟨**header**⟩

and

footer
line position text_string
line position variable format
⋮
end ⟨**footer**⟩

Line, *position*, and *format* have the same meaning as in the print blocks.

The report height (see pg. 46) refers to a number of lines of **print**ed lines. It does not include header or footer text.

This report program

```
report
  header
    1 1 '  Id'
    1 10 Name
    1 60 'Page: '
    1 66 page_number i5
    2 1 ' '
```

```
  end header

  select from techs where position = 'Tech 1'
    print
      1 1 id     i4
      1 10 name  a20
    end print
    mid = id
      print
        2 5 'symbol    date      resistivity'
        3 5 '------    ----      -----------'
      end print
    select from measures where id = mid sort by symbol
      print
        1 5 symbol a8
        1 15 m_date 'yy/mm/dd'
        1 25 resistivity f6.2
      end print
    end select
    print
      1 1 ' '
    end print
  end select
end report
```

produces the following report

```
Id     Name                                              Page:     1

 5     John Jones

   symbol   date      resistivity
   ------   ----      -----------
   Cu        88/01/21   11.90
   Cu        88/01/21   12.40
   Fe        88/03/04   19.40

35     Joe Jackson

   symbol   date      resistivity
   ------   ----      -----------
   Al        88/02/10   13.00
   Al        88/02/10   14.30
```

if

You may want to vary the content or format of your report based on actual data in table columns (or variables). To do so, use **if** statements.

if *test*
— statements executed if *test* is true —
else
— statements executed if *test* is false —
end ⟨**if**⟩

where

test is a normal **where** clause (without the "where"). It may contain columns from the current table, variables, and values.

The *test* is evaluated. If it is true, the statements following **if** are executed and the statements following **else** are skipped. If it is false, the statements following **if** are skipped and the statements following **else** are executed. The **else** and following statements are optional.

This report program

```
*(    Rim report of resistance measurements by conductor )
report

  header
    1  5 'Resistance measurements by conductor'
    1 60 'Page: '
    1 66 page_number i5
    2 1 ' '
    3 1 'Symbol    No.    Name         Resistivity'
    4 1 ' '
  end

  select from conductors
    print
      1 1 symbol a8
      1 10 at_no i4
      1 16 name a15
      1 32 resistivity f6.2
    end print
    xsym 8 = symbol
    mline = 0
    mrest real = 0

    rem  select all measurements for this conductor

    select from measures where symbol = xsym
      mline = mline + 1
      if mline = 1  *(print the sub-header)
        print
          2 5 '   id   date               resistivity'
          3 5 '-----  --------           -----------'
        end print
      end if
      print
        1 5 id i5
        1 12 m_date 'yy/mm/dd'
        1 32 resistivity f6.2
```

```
      end print
      mrest = mrest + resistivity
    end select

    rem print a summary for each conductor

    if mline > 0
      mrest = mrest / mline
      print
        2 10 'Measured resistivity = '
        2 32 mrest f6.2
        3 1 ' '          *( one blank line following summary)
      end print
    else
      print
        2 10 'No measurements taken'
        3 1 ' '
      end print
    end if

  end select

end report
```

prints the following report

```
    Resistance measurements by conductor                  Page:     1

Symbol    No.    Name         Resistivity

Cu          29  Copper            0.12

       id   date               resistivity
    -----  --------            -----------
        5  88/01/21             11.90
        5  88/01/21             12.40
       22  88/02/22             12.50

         Measured resistivity = 12.27

Al          13  Aluminum          0.18

       id   date               resistivity
    -----  --------            -----------
       35  88/02/10             13.00
       35  88/02/10             14.30

         Measured resistivity = 13.65

Fe          26  Iron              0.24

       id   date               resistivity
    -----  --------            -----------
        5  88/03/04             19.40

         Measured resistivity = 19.40

LB        -MV-  Bernstein         -MV-

         No measurements taken

U           92  Uranium           0.44

         No measurements taken
```

while

You have already seen an implicit looping mechanism—the **select** block. Statements inside are executed once for each selected row. The report writer also contains an explicit loop—the **while** block.

while *test*
— statements executed if *test* is true —
end ⟨**while**⟩

where

test is, as with **if**, a normal **where** clause conditions (without the "where"). They may contain columns from the current table, variables, and values.

The *test* is evaluated. If it is 'true', the statements inside the **while** block are executed and the *test* is re-evaluated. As long as *test* is 'true' the statements will continue to be executed. As soon as *test* is evaluated 'false', the **while** block is exited and processing continues with the statement following **end while** statement.

Use the

newpage

statement to end the current page.

This report program, demonstrating the **while** block and the **newpage** statement,

```
*( data entry forms )
report
  select from techs
    print
      1 1 id     i5
      1 10 name  a20
      4 10 'symbol    date    time   resistivity '
      5 1 ' '
    end print

    loop = 1
    while loop <= 5
      print
        1 5 loop i2
        2 10 '------  --------  -----  ----------- '
      end print
      loop = loop + 1
    end while
    newpage
  end select
end report
```

prints the following three-page report.

```
 5    John Jones

      symbol    date    time   resistivity

  1
      ------  --------  -----  -----------
  2
      ------  --------  -----  -----------
  3
      ------  --------  -----  -----------
  4
      ------  --------  -----  -----------
  5
      ------  --------  -----  -----------
```

```
22    Jim Smith

      symbol    date    time   resistivity

  1
      ------  --------  -----  -----------
  2
      ------  --------  -----  -----------
  3
      ------  --------  -----  -----------
  4
      ------  --------  -----  -----------
  5
      ------  --------  -----  -----------
```

```
35    Joe Jackson

      symbol    date    time   resistivity

  1
      ------  --------  -----  -----------
  2
      ------  --------  -----  -----------
  3
      ------  --------  -----  -----------
  4
      ------  --------  -----  -----------
  5
      ------  --------  -----  -----------
```

A procedure allows you to define and invoke a group of statements by name.

procedure *name* ⟨**using** *table*⟩
⋮
end ⟨**procedure**⟩

defines *name* as a procedure. Whenever the statement

name

is encountered, Rim will execute all the statements of the procedure. A procedure cannot have the same name as a variable.

If a procedure contains column names from a table, that table must be identified by the **using** clause.

This example defines a procedure called **spacer**, which prints a blank line after a paragraphed item. It is an alternate form of an earlier example and its output is shown on page 62.

```
report
  procedure spacer   *(define spacer procedure)
    print            *(prints a blank line)
      1 1 ' '
    end print
  end procedure
  select from notes sort by id
    print
      1 5 id     i2
      2 5 symbol a8
      1 15 m_date 'yy/mm/dd'
      2 15 m_time 'hh:mm'
      1 25 notes   a20
```

```
    end print
  spacer      *(invoke spacer procedure)
  end select
end report
```

The report writer allows you to alter some of Rim's parameters during execution of the report program. Use the **set** statement, which is identical to the **set** command. The statements you may use are

set case **ignore** / **respect**

set **input** / **output** *filename*

set report **width** / **height** *count*

which all have the same functionality as the analogous Rim commands. Remember they take effect when they are executed, not when they are compiled.

run

The report writing programs described in previous sections can be given added flexibility through the use of macros (chapter 4).

Consider the example in section 5.10 on page 71. Suppose you wanted to generalize this report to print an arbitrary number of lines per person, and to report for only selected persons.

One way to do this would be to use two macros: **_count** and **_where**. Then change line 3 of the report to

```
select from techs _where
```

and change line 12 to

```
while loop <_count
```

Define these macros before running the report. For example,

```
macro _count = 10; macro _where = ' '
input forms
```

or

```
macro _count = 7;
macro _where = 'where position = "Tech 1"'
input forms
```

You can make this operation much more friendly through the use of this **run**

macro.[1]

```
macro run = +
    'macro arg_1 = "' 2 '";' +
    'macro arg_2 = "' 3 '";' +
    'macro arg_3 = "' 4 '";' +
    'input ' 1
```

Here is what happens when you type

```
run forms 5 'where position like "Tech*"'
```

1. The macro **arg_1** is defined to be '5'.
2. The macro **arg_2** is defined to be '`where position like "Tech*"`'.
3. The macro **arg_3** is undefined.
4. The file 'forms'[2] is input.

Of course, the report program on file **forms** must be aware of this **run** invocation. It should be written as follows.

```
*( data entry forms )
*( use is:  RUN filename count where_clause )

report
  select from techs   arg_2
    print
      1 1 id     i5
      1 10 name  a20
      4 10 'symbol    date    time   resistivity '
      5 1 ' '
    end print
```

[1] You might want to put this macro definition in your login or database initialization files.

[2] Some systems may add an extension to this name.

```
    loop = 1
    while loop <=  arg_1
      print
        1 5 loop i2
        2 10 '------  --------  -----  ----------- '
      end print
      loop = loop + 1
    end while
    newpage
  end select
end report
```

to produce the output shown on page 71.

Chapter 6

Using Rim with a programming language

This chapter described the programming language interface available to Rim users. Rim is written in fortran-77, and the descriptions in this chapter use fortran-77 terminology. However, you may invoke Rim functions from any language that can call fortran-77 function subroutines.

This interface is very simple—there is a function subroutine to execute Rim commands, and another function subroutine to transfer data.

When Rim detects an error it posts the error number in the integer variable **RMSTAT**, the only element of the **/RIMCOM/** common block.

```
COMMON /RIMCOM/ RMSTAT
INTEGER RMSTAT
```

If no error occurs, **RMSTAT** will be zero. Table 6.1 lists the **RMSTAT** error codes. Table 6.2 lists **RMSTAT** error codes that indicate Rim internal errors and should not be encountered. Normally a status of 0 is good, a status greater than 0 is bad, and a negative status means there are no more rows to retrieve.

Programs may interrupt Rim's processing by setting the integer variable **HXFLAG**, the only element of the **/RIMSTP/** common block.

```
COMMON /RIMSTP/ HXFLAG
INTEGER HXFLAG
```

Users of the program interface may independently access several tables simultaneously. An indexing integer specifies which table is to be accessed. The index is ignored for those commands which do not access a specific table (e.g. **open**). The index has a range of 1–10 (ZPIMAX).

The logical function

RIM(*index*, *command*)

where

index is the integer table index,
command is a string containing a Rim command.

returns a value of true if the command is executed successfully, and false if there has been an error. In the latter case **RMSTAT** will contain the error number.

For example, to open the **Sample** database and enter the user password of **metals**[1] you can include

[1] Our Sample database as described in this reference had no password.

-1	no more data available for retrieval
0	ok–operation successful
1	table not found
2	no database is open
3	column not found
4	syntax error in command
5	table already exists
6	HXFLAG interrupt
7	invalid name
8	no authority for operation
9	no permission on a table
10	files do not contain a Rim database
12	files incompletely updated (not fatal)
13	database is attached in read only mode
14	database is being updated
15	tuple too long
16	database has not been opened
20	undefined table
30	undefined column
40	where clause too complex
42	unrecognized comparison operator
43	'like' only available for text columns
45	unrecognized logical operator
46	compared columns must be the same type/length
47	lists are valid only for eq and ne
50	**select** not called
60	**get** not called
70	multiple table index out of range
80	variable length columns may not be sorted
81	the number of sorted columns is too large
89	sort system error
90	unauthorized table access
91	table already exists
92	bad column type
93	bad column length
94	too many or two few columns
95	row too big to define
100	illegal variable length row definition (load/put)

Figure 6.1: `RMSTAT` error codes

1001	buffer size problem – BLKCHG,BLKDEF
1002	undefined block – BLKLOC
1003	cannot find a larger b-tree value – BTADD,PUTDAT
1004	cannot find b-tree block – BTPUT
21xx	random file error xx on file1
22xx	random file error xx on file2
23xx	random file error xx on file3
3000	sort error - no buffer available
31xx	sort error - xx on file open

Figure 6.2: RMSTAT error codes indicating Rim internal errors.

```
IF (.NOT.RIM(0,'open sample')) GOTO  <error label>
IF (.NOT.RIM(0,'user metals')) <ditto>
```

in your fortran program.

The logical function

RIMDM(*index*, *command*, *tuple*)

where

index is the integer table index,
command is a string containing a Rim data-movement command, and
tuple is the integer array containing the data to be transferred.

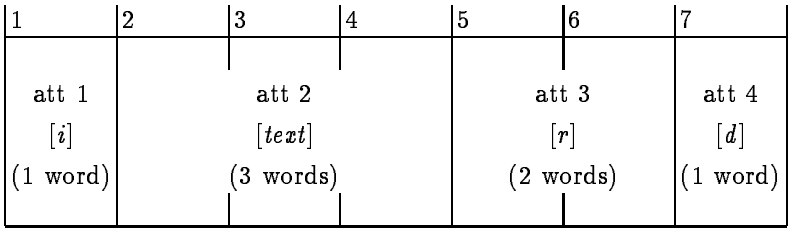
returns a value of 'true' if the command is executed successfully, and 'false' if there has been an error. In the latter case **RMSTAT** will contain the

error number. You must have executed a Rim 'table-selection' command (presently only **select** and **load**) before issuing a data-movement command.

The data-movement commands are:

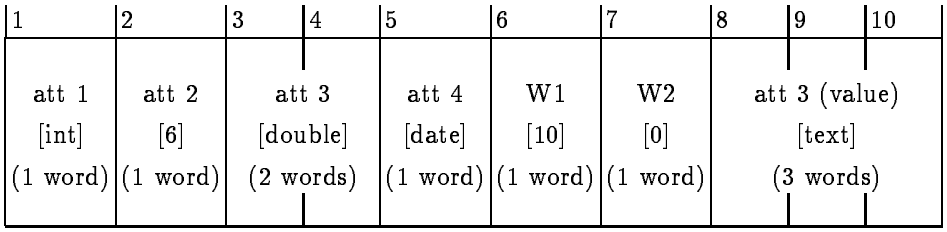
Command	Description	Prerequisite
get	Retrieves the next row from the table.	**select** (**RIM**)
put	Replaces the current row into the table.	**get** (**RIMDM**)
del	Deletes the current row from the table.	**get** (**RIMDM**)
load	Loads a new row at the end of the current table.	**load** (**RIM**)

The word 'tuple' is used to mean an array containing a row of table data. Most columns are located sequentially in the tuple and occupy a fixed number of words depending on the column's type. Integers occupy one word per number. Double precision numbers occupy two words per number. Text is packed CPW characters per word, where CPW is a machine dependent parameter. On 32-bit machines CPW = 4. We will assume the value of 4 in this discussion. A tuple containing an integer (value i), a 10-character text item (value *text*), a double (value r), and a date (value d) occupies 7 words and looks like this:



Variable length columns are the exception to this rule. They have a one-word pointer at their position in the tuple. This pointer contains the offset (from

the start of the tuple) to the actual location of the variable column's data. At that offset a two word header contains length information. Following this is the actual value. The same tuple, if the character string is variable length, occupies 10 words and looks like this:



The variable length header (words W1 and W2 in the example) contains

Data type	W1	W2
text	# chars	0
vector	# cols	0
matrix	# rows	#cols

Data in text columns are represented by integer values. Before you can use these in your program you must convert them to characters in character variables. The subroutine

CALL STRASC(*string*, *tuple* (*pos*), *len*)

converts a text column at *tuple*(pos) of length *len* characters into a character string in *string*, type CHARACTER.

To move characters back into the tuple use the inverse subroutine

CALL ASCTXT(*tuple*(*pos*)**,** *len*, *string*)

which moves the characters in *string* into the tuple at *pos*, length *len*.

Dates are represented by integer Julian values, according to the algorithm of Tantzen.[2] Use the subroutine

CALL DATJUL(*day*, *month*, *year*, *Julian*)

to convert a Julian integer into day, month, and year integers.

Use the inverse subroutine

CALL JULDAT(*day*, *month*, *year*, *Julian*)

to convert the day, month, and year into a Julian integer.

Times are also represented by integer values, according to the formula

$$\text{time integer} = \text{hours} * 3600 + \text{minutes} * 60 + \text{seconds}.$$

There are no subroutines provided to facilitate conversion.

A "missing value" is indicated by the integer value ZIMISS (2147483647) and a "not applicable" value is indicated by the integer value ZINAPP (2147483646).

[2] Collected Algorithms of the ACM, 199, R. Tantzen.

Open the database and set the password, if one is required, and then issue the select command

select from *table*
⟨**where** *where clause*⟩
⟨**sort** ⟨**by**⟩ *sort clause*⟩

via the **RIM** function. No column list is allowed as Rim always transfers complete rows.

Now you can **get** rows from this table, modify them and **put** them back, or **del**ete them. A prototype retrieval loop is illustrated in Figure 6.3. Note that the character variables must be unpacked from the integer tuple array with **STRASC** and must be replaced with **ASCTXT**.

Open the database and set the password, if one is required, and then issue the load command

load *table*

via the **RIM** function.

Now you can **load** rows into this table with the **RIMDM** function.

```
      ⋮
      IF (.NOT.RIM(1,'open example')) GOTO error
      ⋮
      IF (.NOT.RIM(1,
     1 'select from conductors sort by symbol')) GOTO error

label IF (RIMDM(1,'get',TUPLE)) THEN
         ⋮
         IF (modifying) THEN
            IF (.NOT.RIMDM(1,'put',TUPLE)) GOTO error
         ELSE IF (deleting) THEN
            IF (.NOT.RIMDM(1,'del',TUPLE)) GOTO error
         ENDIF
         GOTO label
      ELSE
         IF (RMSTAT.NE.-1) GOTO error
      ENDIF ⋮
```

Figure 6.3: Sample data retrieval with update

Appendix A

Using Rim at UCS

This chapter tells you how to use Rim on the various UCS computers.

Start Rim with the command

    % **rim**

If the initialization file **.rimrc** exists in your home directory, it will be input when Rim starts.

You can also run Rim in a batch mode by specifying an input file. To do this start Rim with the command

% **rim** *input_file*

Rim will process the *input_file* and will not communicate with your terminal.

Filenames, and the database name, may include directory information. For example,

open 'user/elsewhere/hisdb'

opens the database **hisdb** on the directory **user/elsewhere**. The name must be enclosed in quotes only if it contains special characters.

UNIX Rim does not assume any extensions for either input or output files. Database files have the extensions **.rimdb1**, **.rimdb2**, and **.rimdb3**.

To load fortran programs with the Rim interface include the **rimlib.a** library with the **-lrim** option in your **ld** or **f77** command.

UNIX Rim allows you to execute shell commands by

system *command*

where *command* will normally be a quoted string. Rim closes your database prior to executing the system command so your data should be OK even if you cannot return.

Start Rim with the command

$ **rim**

If the initialization file **login.rim** exists in your login directory, it will be input when Rim starts.

You can also run Rim in a batch mode by specifying an input file. To do this start Rim with the command

$ **rim** *input_file*

Rim will process the *input_file* and will not communicate with your terminal.

Filenames, and the database name, may include directory information. For example,

open '[zz99.elsewhere]hisdb'

opens the database **hisdb** on the directory [**zz99.elsewhere**]. The name must be enclosed in quotes only if it contains special characters.

VMS Rim assumes an extension of **.rim** for input files and **.lis** for output files. Database files have the extensions **.rimdb1**, **.rimdb2**, and **.rimdb3**.

To load fortran programs with the Rim interface, include the library

$$rim:rimlib/lib

in your link command.

Before running Rim you must run **setup** to access the appropriate disks.

setup rim

Start Rim with the command

rim

If the initialization file **profile rim** exists, it will be input when Rim starts.

You can also run Rim in a batch mode by specifying an input file. To do this start Rim with the command

rim *input_file*

Rim will process the *input_file* and will not communicate with your terminal.

Then start Rim with the command

rim *input_file*

and Rim will read the *input_file* if it is given. *input_file* may contain a type and mode.

You use **setup** only once per session. It may be included in your **profile exec** file.

Filenames may include types and modes and may use spaces or periods as delimiters.

'file type m' and **file.type.m**

are equivalent.

The database specification may include a mode letter. For example,

open hisdb c

opens the database **hisdb** on the disk **c**.

CMS Rim assumes a file type of **rim** for input files and **rimlist** for output files. Database files have the types **rimdb1**, **rimdb2**, and **rimdb3**.

To load fortran programs with the Rim interface, include the libraries RIMLIB and UTILITY in your GLOBAL TXTLIB command. RIMLIB and this sample loader EXEC (RIMLOAD EXEC) may be found on the Rim disk.

```
/* load program using rimlib */
arg fname "(" options
"GLOBAL TXTLIB RIMLIB VLNKMLIB VFORTLIB CMSLIB UTILITY"
"LOAD" fname "(" options
exit
```

CMS Rim allows you to execute selected CMS or CP commands by

system *command*

where *command* will normally be a quoted string. You can run XEDIT and most EXECs, but you cannot run other programs. Rim closes your database prior to executing the system command so your data should be OK even if you cannot return.

Since Rim will read from the program stack you can write 'Rim EXECs' which build and execute Rim commands. One example of this capability is

```
/* Rim exec to load notes entries */
say "Enter the symbol and your id"
parse pull symbol id
if id = '' then exit
D = date(0)
T = substr(time(),1,5)
say "Enter notes"
parse pull notes
queue "load notes"
queue symbol id "'"D"' '"T"' +"
queue "'"notes"'"
queue "end"
queue "system 'exec l_notes'"
exit
```

Figure A.1: Sample VM/CMS Rim EXEC
This EXEC, named "L_NOTES EXEC" will interactively load the notes table of the Sample relation.

an interactive data loader for the **notes** table. Figure A.1 shows the EXEC which reads the data, formats a load command, and returns it to Rim for execution. It also stacks a command to re-execute itself unless it has been given a null response.

Appendix B

Distribution and License

Rim is free; this means that everyone is free to use it and free to redistribute it on a free basis. Rim is not in the public domain; it is copyrighted and there are restrictions on its distribution—restrictions similar to those of GNU software.

The easiest way to get a copy of Rim is from someone else who has it. You need not need permission. If you cannot get a copy this way, you can order one from University Computing Services. Though Rim itself is free, our distribution service is not. For further information, contact

University of Washington
University Computing Services, HG–45
3737 Brooklyn Ave NE
Seattle, WA 98105
USA

Out intention is to give everyone the right to share Rim. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

For our own protection, we must make certain that everyone finds out that there is no warranty for Rim.

1. You may copy and distribute verbatim copies of Rim source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each file a valid copyright notice "Copyright ©1988 University of Washington" (or with whatever year is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of Rim a copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.

2. You may modify your copy or copies of Rim source code or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

   - cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of Rim or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).

- You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another unrelated program with this program (or its derivative) on a volume of a storage or distribution medium does not bring the other program under the scope of these terms.

3. You may copy and distribute Rim (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal shipping charge) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs.

4. You may not copy, sublicense, distribute or transfer Rim except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer Rim is void and your rights to use Rim under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.

Index