

The Relational Data System LIDAS

© (1984) by
Günter Dotzel and Klaus Moritzen

A database system for micro-computers in Modula-2

Introduction:

In former days, databases were used only in a scientific or commercial environment on large mainframes to cope with "mountains of data". But the advantages of a database system, the ability to analyse data in an arbitrary way, involve new applications, like information retrieval, management information systems, computer aided design, bill of material processing, project scheduling, etc. Today we can buy database systems off the shelf for minicomputers, personal workstations or even home computers. Many of them do not use the results of the database research in their design and implementation and in practice they show some serious limitations.

The Lilith Database System (LIDAS) is programmed in Modula-2 and with LIDAS we want to study the concepts of a "state-of-the-art" relational database system. We will further discuss the portation of the LIDAS run time system to a conventional 16-Bit micro- (mini)-computer. With the portation of the database run time system and its procedural interface, the so-called **Relational Element Manager**, a very powerful software tool is now available for PDP-11 under RT-11 operating system.

KEYWORDS: Modula-2, Relational Database System, LIDAS, Modula/R.

The LIDAS-System and Modula/R:

Modula/R is a database programming language that extends Modula-2 by a few constructs, namely the data structure RELATION, predicate-oriented selection of relation elements and a transaction concept [REIM83a]. Modula/R is part of the LIDAS-System. With LIDAS, database programming can be performed either using Modula/R (1) or using Modula-2 [WIRT83] with the procedural database interface (2):

(1) Modula/R is a database language intended for end-user programming. It provides a total integration of the database concepts into the language. Modula/R supports comfortable and reliable

design of database applications. *At compile-time* the Modula/R compiler translates a program written in Modula/R using the database specification found in the user database definition module (*DDM*). The objects defined in the DDM are permanent and will survive the program's life. The description of these objects is kept in the user database file(s).

In Modula/R programs one can formulate boolean expressions (i.e predicates) of the power of the first-order calculus. The Modula/R compiler generates set operations on relations, handled by the so-called Query Evaluation Manager (*QEM*) *at run-time*. The QEM transforms a normalized relational expression into a sequence of operations on single elements of relations provided by RDS.

(2) The procedural element interface of the *Relational Data System (RDS)*, the so-called *Relational Element Manager (RDSREM)* allows programming of system utilities and dedicated database applications.

The method (2) for database programming in Modula-2, together with the portation challenge of RDS and database utility programs are described here. Furthermore, the highlights of a modern database system are shown. The database performance measurement gives an impression of the capabilities of RDS, without the overhead of QEM. For more informations on method (1) and Modula/R see [REIM83b].

The *Lilith Database System (LIDAS)* is programmed entirely in Modula-2 and is available from ETH-Zürich since June 1983 [ZEHN83]. The distribution kit includes a relational database run time system (modules RDS*), RDINIT: a utility program for interactive definition and modification of the database description, DIALOG: a utility program for operating the one-tuple interface RDSREM, DBA (Database Administration) for saving (restoring) a database in (from) a sequential file for backup purposes and access path reorganisation, the *query evaluation manager* modules (QEM*), and the Modula/R compiler, based upon the four pass Modula-2 (C19) compiler, producing M-code for the Lilith (Modula Machine) and generating calls for the Modula/R run-time system QEM.

Some other systems based on LIDAS are in field test at the ETH-Zürich, namely GAMBIT, DISCUSS and a distributed database system [ZEHN83].

GAMBIT is an interactive graphical database design tool using bit-map screen and mouse features. It supports the design of database structures, integrity constraints and transactions. GAMBIT produces Modula/R programs: a database definition module and code for transaction procedures.

HIQUEL is an interactive query language, used to define and use hierarchies. HIQUEL is part of DISCUSS, a database interface specified for a casual user of a small system.

Database services are provided for personal computers connected by the local area network MAGNET (*Modula oriented general purpose network*) by a Federative Database Server System [DIEN83].

LIDAS shows that Modula-2 is an adequate tool to implement large system programs by decomposing them into small modules. The LIDAS implementation for Lilith is properly structured and defines different abstraction levels, together about 50 modules. A powerful and efficient database concept was realized, satisfying the requirements of casual users, parametric users and database designers. An interactive database definition program allows to define and extend an arbitrary number of relations and their interconnection. Generalized B^{*}-trees offer a fast access to specified relation elements. It fully supports atomic transactions.

LIDAS is distributed in source form only, but is neither intended nor *ready for use under PDP-11*. Portation requires a lot of work because of the following reasons: Modula-2 is a portable language, only if the system dependent module SYSTEM is not used (imported). The RDS is a system dependent component. In fact, almost every module of the RDS (several thousand lines of Modula-2 source code) import the type ADDRESS, WORD and the function procedure ADR from the pseudo module SYSTEM. This involves portation problems on hardware utilizing a different addressing scheme. The Lilith is a word addressing machine. The PDP-11 is a byte addressing machine.

Another problem is the extensive use of special operating system specific features: the file system of MEDOS-2 (operating system of Lilith) and the very simple, but efficient file system of RT-11 are not compatible. Files are contiguous in RT-11, in MEDOS-2 they aren't. The file name syntax is different.

Memory Requirements:

Not only the address calculation (word versus byte machine) is affected: the RDS modules need a lot of memory space for code and data (buffer pool and heap). The directly addressable memory is maximal 64KByte on PDP-11 without memory management unit (MMU) and 64KWord on Lilith.

The Lilith has a more compact machine code, since it directly executes the M-code, produced by the four pass Lilith Modula-2 compiler. The M-code was defined to support the language Modula-2 in an optimal way. The PDP-11 machine code produced by the five pass PDP-11 Modula-2 compiler is about two times larger (run time checks disabled). So the RDS implementation for PDP-11 has to be significantly optimized in size, to leave enough space for application programs. In the following we will discuss the portation challenge in detail:

The RDS needs 17KByte on Lilith without the page buffer of the buffer manager [ZEHN83], [REBS83a]. The correspondent PDP-11 implementation needs 28KBytes of code (all run time checks enabled). There is enough memory available, to operate the LIDAS utilities RDINIT, DIALOG and DBA on PDP-11 under the standard Modula-2 environment. For the QEM (and for Modula/R programs) the Modula-2/XM environment (M2XM), described in [DOTZ84], must be used. M2XM allows programs up to 2MByte in size to be executed on a PDP-11 with MMU (e.g. PDP-11/23, PDP-11/73, Professional 350 running RT-11SJ V05.1). The QEM is about 40KByte of code linked atop of the RDS and can be compiled using M2XM only, since the QEM source modules are very large and need a lot of stack and heap (about 20KByte) at compile time.

Database Portation Considerations:

The two main portation problems are considered here: the different addressing schemes and the incompatible file systems.

Word versus Byte Addressing:

Some modules or module parts are written to be independent from a word or byte addressing scheme. These modules need only simple modifications.

(1) Hardware independency can be realized for example using the following technique:

TYPE

```
Page = ARRAY [0 .. 255] OF CARDINAL; (* 256 WORD's per physical disk block *)
(** assume TSIZE (WORD) = TSIZE (CARDINAL) **)
pw = POINTER TO Page;
```

VAR q: pw;

```
w, index: CARDINAL;
```

Processing a data page is *hardware independent* if you use:

```
INC (index); w := q^ [index];
```

to access the next word in a page; where q may be initialized, for example, by the procedure `RDSBUM.GetPage`, which delivers the starting address a of a data page with page number p in the buffer pool.

The definition of `GetPage` is shown here. It serves as a relevant example for a procedure of the buffer manager `RDSBUM`, which implements the lowest level of the RDS:

```
PROCEDURE GetPage (p: PageNo; pk: PageKind; VAR a: ADDRESS);
```

```
(* The page  $p$  is transferred from the database file, if it is not already in the buffer pool and
the starting address of page  $p$  is assigned to  $a$ . *)
```

(2) The following *hardware dependent*, in fact, a little more efficient method, to access the next word in memory, was used in some modules, which need serious modifications. This method should not be used (assume q is initialized like above):

```
VAR q: ADDRESS; (* compatible to the type POINTER TO WORD *)
```

```
w: CARDINAL; (** assume TSIZE (WORD) = TSIZE (CARDINAL) **)
```

```
INC (q); w := CARDINAL ( q^ ); (* correct on Lilith: TSIZE(CARDINAL)=1 *)
```

```
INC (q, 2); w := CARDINAL ( q^ ); (* correct on PDP-11: TSIZE(CARDINAL)=2 *)
```

(3) Now assume the proper method (1) is used. Further incompatibilities and hardware dependencies (of 2nd degree) have to be eliminated. The following statement has to be examined, if the value $q^{\text{[index]}}$ is used to specify a counter, an offset or the length of an object, for example.

```
INC ( q^ [index] ); (* Lilith implementation *)
```

If it is later used as an address or offset, the statement has to be modified:

```
INC ( q^ [index], 2 ); (* correct on PDP-11 *)
```

Different File Systems:

The buffer manager was significantly reduced in size and improved in performance, since the RT-11 file system allows direct random DMA-transfers from and to a file. The buffer manager is responsible for the buffer pool and page management. The page number is directly used to specify the block number for the read and write requests for the database file(s) and the UNDO log file(s). The recovery feature implementation was modified, since the relevant length of the current UNDO file is not available from the RT-11 directory system after a system crash.

Further the modules, which use the FileSystem or the I/O-stream handler (e.g. for generating files, saving and restoring the database), have to be changed: RDINIT, DIALOG and DBA. For the save (restore) module of DBA a new module *RDStreams* was developed for sequential writing (reading) relations. The number of relations defined by a database is determined before writing the element data of the different relations, to avoid stream repositioning at the end of the saving process. The length of an element is stored by *RDStreams* automatically before the writing of individual element data. This length is compared to the requested length at recovery time and the I/O-result can be examined by the function procedure *RDStreams.Done*.

Highlights of the Relational Database System RDS:

Let's have a closer look at the main features of the LIDAS run time system. The RT-11 specific topics, especially the file types (where file type means file name extension) used, are also discussed here.

(a) Database Files, Data Description and Keys:

A database consists of maximal 86 data files (together maximal 16MByte) containing the database description, the access path information and the element data.

The database DBX for example consists of the RT-11 files DBX.D01, DBX.D02, up to DBX.D86 on a user specified device. The size of each .Dxx file (currently 384 blocks each) is defined by the buffer management module.

The description consists of the relation names, the attribute types of each relation, the identification key name together with its associated attribute types and the description of *further non-identifying keys* (multi-key feature).

A key is defined by one or more attributes (fields) of a relation, specified in logical order. An attribute is of CARDINAL, INTEGER, REAL, CHAR or BOOLEAN type or an ARRAY OF CHAR (string type, up to 80 characters).

The description is generated by the interactive program RDINIT.

(b) *Recovery:*

The recovery mechanism uses the file type .Rxx with 384 blocks each on logical device RD for the UNDO LOG: DBX.R01, DBX.R02, ... in our example.

(c) *Temporary Data:*

Temporary data which need not to be recovered after a system crash, is used by the QEM modules only and is stored in files of the type .Txx with 384 blocks each: DBX.T01, DBX.T02, ... in our example.

(d) *Database Saving and Restoring:*

A database could be saved (restored) using the program DBA, which writes (reads) sequential files of the type .Sxx on (from) a user specified device (default device RS): DBX.S01, DBX.S02, ... in our example. The file length is up to 384 blocks each.

The sequential data stream consists of a header, specifying the number of relations, the name of a relation, the cardinality of a relation (number of elements) the length of a relation element and the element data.

(e) *Access Path Structure and Programming Example:*

The *generalized B^{*}*-tree concept provides fast access to relation elements. More than one relation can be supported by one tree if there are relations with the same [identification] key attribute combination. Such a tree contains important interrelational information and allows *fast join operations* for database relations. The module RDSACM (access manager) implements the so-called *generalized access path structure* [HAER78] (It is assumed, the reader is familiar with some database techniques).

Database Definition Example:

To demonstrate this feature, let's examine the following simple database definition, where the *identifying* key is underlined and upper-case letters are used for the key attributes supported by the same physical access path (tree). The names specified in brackets can be regarded as field names in a record data type:

```
PersonType = (PersNo, PersKind, LastName, FirstName, Title);
AddressType = (PERSNO, ADRNO, AdrType, Country, Zip, City);
CompanyAdrType = (PERSNO, ADRNO, Line1, Line2, Line3, Line4);
TelefonType = (PERSNO, ADRNO, TelNr);
KeywordType = (KeywordNo, KeywordText);
InterestType = (PERSNO, ADRNO, KeywordNo, Priority);
```

Database Programming Example:

A simple programming example in Modula-2 shows how to obtain all informations related to a given *LastName* in the relation *Person*, stored in the database; where all persons with last name "**Roberts**", at address number 0, should be examined (e.g. printed).

The RDSREM interface, further discussed below, is used here. The procedure RDSREM. *Obtain* looks for an element of a relation *r*, using the access mode *m* and the access path *k*, and copies the element to the variable (record structure) referenced by *re*. The types *Relation* and *Key* are **HIDDEN**.

```
PROCEDURE Obtain (r: Relation; re: ADDRESS; k: Key; m: Mode);
```

If *Obtain* succeeds, the boolean procedure *Done* returns TRUE and the element retrieved can be processed. If the requested element is not available, *Done* returns FALSE.

The Modula-2 data type declaration part is omitted, because self-explanatory names are used in the database definition. Only the variable definition is shown here:

VAR

```
Person: PersonType; Address: AddressType; CompanyAdr: CompanyAdrType;
Telefon: TelefonType; Interest, Keyword: InterestType;
```

```
RelPerson, RelAddress, RelCompanyAdr, RelTelefon, RelInterest, RelKeyword:
Relation;
```

```
KeyPerson, KeyAddress, KeyCompanyAdr, KeyTelefon, KeyInterest, KeyKeyword:
Key;
```

```
(* ... assume proper calls of InitRelation and InitKey.
   In InitKey the identifying key is used if not stated otherwise.
   Done() = TRUE means last command successfully executed. *)
```

```
Person.LastName := "Roberts";
```

```
(*
   Initialize key-value "Roberts" to data record Person.
```

```
KeyPerson is a non-identifying key and selects LastName.
```

```
Give this record to procedure Obtain to fill in the other fields:
```

```
*)
```

```

Obtain (RelPerson, ADR(Person), KeyPerson, firstkey);
(*)
  If the relation RelPerson contains matching elements, the information
  of the first element (firstkey) is copied to the record Person.
*)
WHILE Done() DO
  (* obtain all informations of all persons with the same LastName *)

  WITH CompanyAdr DO PersNo := Person.PersNo; AdrNo := 0; END;
  Obtain (RelCompanyAdr, ADR(CompanyAdr), KeyCompanyAdr, firstkey);

  WITH Address DO PersNo := Person.PersNo; AdrNo := 0; END;
  Obtain (RelAddress, ADR(Address), KeyAddress, firstkey);

  WITH Telefon DO PersNo := Person.PersNo; AdrNo := 0; END;

  (* (non-identifying) KeyTelefon is (PersNo, AdrNo) *)
  Obtain (RelTelefon, ADR(Telefon), KeyTelefon, firstkey);
  WHILE Done() DO (* get all phone numbers of this person: *)
    Obtain (RelTelefon, ADR(Telefon), KeyTelefon, nextequal);
  END;

  WITH Interest DO PersNo := Person.PersNo; AdrNo := 0; END;

  (* (non-identifying) KeyInterest is (PersNo, AdrNo) *)
  Obtain (RelInterest, ADR(Interest), KeyInterest, firstkey);
  WHILE Done() DO (* get all interests of this person: *)
    WITH Keyword DO KeywordNo := Interest.KeywordNo; END;
    Obtain (RelKeyword, ADR(Keyword), KeyKeyword, firstkey);

    Obtain (RelInterest, ADR(Interest), KeyInterest, nextequal);
  END;

  Obtain (RelPerson, ADR(Person), KeyPerson, nextequal);
END; (* WHILE Done() *)

```

Generalized Access Path Effect:

The keys **KeyAddress**, **KeyCompanyAdr**, **KeyTelefon**, **KeyInterest** are all of the same attribute combination (*PERSNO, ADRNO*). This key is not necessarily an identification key (see relation types **TelefonType** and **InterestType**). The execution of the procedure *Obtain* loads the selected user data record into **Person**, **Address**, **CompanyAdr**, **Telefon**, **Interest**, or **Keyword** and requires at maximum *one disk access each*.

(f) *RDS Module Structure and Software Hierarchy:*

Here we have a short look at the RDS module structure. It is fully explained in [ZEHN83]. The different levels are marked by "(Lx)".

On top of the RDS we have the module RDSREM (relational element interface) which serves as a procedural database interface (L4).

The modules RDSACM (access manager) and RDSMAM (data manager) implement the next layer (L3).

RDSACM uses the tree page handler consisting of the modules RDSNLP (non-leaf-page handler) and RDSLPH (leaf-page handler) (L3.1) and is responsible for the maintenance of the access path.

RDS DAM implements the tuple identifier (TID) concept (L3.2). A tuple identifier consists of two words: a page number, specifying a unique data page and a page offset, specifying an address within the data page.

The module RDS DEG (description generator, L3.3) is only used by the interactive program RDINIT and the design tool GAMBIT, which have to be considered as database definition or modification utilities.

The module RDS DEM (description manager) forms the next level (L2) and is a central component for managing the database description lists.

On the lowest RDS level (L1) we have the module RDS BUM (buffer manager), which makes use of the underlying (operating system specific) file system. It provides operations for reading and writing pages with a fixed length (256 words) from and to the database files. It handles the following page operations: fixing in buffer pool, releasing, marking for modification, returning and requesting. Again the procedure *GetPage* serves as an example and is described here in detail:

```
PROCEDURE GetPage (p: PageNo; pk: PageKind; VAR a: ADDRESS);
```

```
(* Look for the page with number p in buffer pool. If p not found, find a free slot in the
buffer frame (if there is no free slot available, locate a "released" page which has the lowest
priority, using the "generalized second chance (generalized clock) algorithm" from
[SMIT78], clock out the page, if it was "modified", i.e. it was marked for modification
previously) and read the page p from the database file into the free slot. The starting
address of page p is assigned to a. A page is either of system, user or temporary type. The
page type determines its priority. *)
```

RDS BUM also provides transaction support via *Begin*-, *Commit*- and *AbortTransaction*.

(g) *The Relational Element Interface:*

The element interface between the run time system RDS and the high level components of LIDAS is called RDS REM (Relation Element Manager) and provides procedures for opening and closing of

- 1) the database: *OpenDB*, *CloseDB*;
- 2) relations: *InitRelation*;
- 3) access paths: *InitKey*.

The data managing procedures provided are *Find*, *Obtain*, *Insert*, *Delete* and *Replace*.

The consistency-preserving operations, which support the atomicity, are *BeginTransaction* and *CommitTransaction*.

The access modes for the *Find* and *Obtain* operation are:

(first, last, next, prior, nextequal, priorequal, firstkey, lastkey, keyornext, keyorprior, current)

The status of a relation or previous executed operation can be checked by *Done*.

The module RDS REM is the procedural programming interface for Modula-2. Though it is the highest RDS level, its definition is only visible for system programmers and database experts and serves as a common data handling facility for all LIDAS user interfaces. The definition module was extended by the description found in [REBS83b] and is listed in the appendix.

Database Performance:

The performance of the run time system RDS* can be examined using the utility DBA for saving and restoring a database. DBA is linked to the following modules: DBASAV, RDSREM, RDSACM, RDSNLP, RDSLPH, RSDAM, RSDDEM, RDSBUM, heap manager *Storage* and the file system *Files*.

The saving process is very fast. This operation is required very often, to generate a sequential file for backup purposes. The restoring process is slower, since access path manipulations and UNDO LOG operations have to be performed at every *Insert* operation. *CommitTransaction* is executed after a fixed number of *Inserts*. A fast I/O-device should be used on restoration (a hard disk or even better a solid-state [RAM-] disk). The restoration of the author's mailing list database consisting of four relations (about 950 persons, addresses and phone numbers) takes only about five minutes on a PDP-11/23 with 1MByte of extended memory (VM), if the database is kept in VM for the restoration process. Currently the buffer pool size is only 10 pages (40 pages on Lilith [REBS83a]).

Portation State and Future Developments:

The components of RDS and the utilities were modified and implemented on PDP-11/RT-11. The system is in production use since Februar 1984. A minimal system configuration consisting of 64KBytes main storage and 2MBytes of mass storage (floppy disk) is required for the LIDAS utilities and small database application programs atop of the RDS; this is actually the configuration used for the portation of RDS. At least 128KBytes and a PDP-11 with MMU are recommended for large applications under M2XM. The components of the Modula/R compiler (pass 4) for PDP-11 and the Modula/R run-time system QEM are under development.

Further implementation effort will be made to make the system work under Share-11 [HAMM83]. Share-11 is an efficient multi-user upgrade for RT-11, which allows the RT-11 and TSX compatible execution of several parallel jobs with a *resident job space* of up to 62KByte each. Share-11 is faster than other multi-user systems like TSX, RSX, UNIX, and it maintains the full comfort of the single-user operating system RT-11.

Acknowledgement:

The author would like to thank the Institut für Informatik, ETH-Zürich for the proper preparation of the documentation and the release kit of LIDAS. Thanks are due to Andreas Diener and Manuel Reimer for their comments and Hans-Georg Daun for proofreading.

REFERENCES:

- [DIEN83]: Diener, Andreas; Brägger, Richard; Dudler, Andreas; Zehnder, Carl August: **Database Services for Personal Computers linked by a Local Area Network**, Proceedings of the ACM SIGSMALL - SIGPC Conference on Small and Personal Computers, San Diego CA, U.S.A., Dec. 1983. (Reprinted in ZEHN83).
- [DOTZ84]: Dotzel, Günter; Moritzen, Klaus: **The Programming Language Modula-2 and its Environment**, ModulaWare, Schwalbenweg 12, D-8520 Erlangen, F.R.G., Mar. 1984.

- [HAER78]: Haerder, T.: **Implementing a Generalized Access Path Structure for a Relational Database System**, ACM TODS, Vol. 3, No. 3, Sep. 1978, pp. 285-298.
- [HAMM83]: Hammond, Ian: **Introducing Share-eleven, HAMMOND-software**, An der Lutter 32, D-3400 Göttingen, F.R.G., 1983.
- [REBS83a]: Rebsamen, Jürg; Reimer, Manuel; Ursprung, Peter; Zehnder, Carl August; Diener, Andreas: **LIDAS - The Database System for the Personal Computer Lilith**, in [ZEHN83], pp. 21-47.
- [REBS83b]: Rebsamen, Jürg; Reimer, Manuel; Diener, Andreas: **Relation Element Manager RDSREM**, Research Project LIDAS, Institut für Informatik, ETH-Zürich, Jan. 1983.
- [REIM83a]: Reimer, Manuel: **Modula/R Report -Lilith Version-**, Research Project LIDAS, ETH-Zürich, Feb. 1983.
- [REIM83b]: Reimer, Manuel: **Implementation of the Database Programming Language Modula/R on the Personal Computer Lilith**, Research Project LIDAS, ETH-Zürich, Feb. 1983.
- [SMIT78]: Smith, A. J.: **Sequentiality and Prefetching in Database Systems**, ACM TODS, Vol. 3, No. 3, Sep. 1978, pp. 223-247.
- [WIRT83]: Wirth, Niklaus: **Programming in Modula-2**, Springer Verlag, 1982, 1983 (2nd. ed.).
- [ZEHN83]: Zehnder, Carl August (Ed.): **Database Techniques for Professional Workstations**, Institut für Informatik, ETH-Zürich, CH-8092 Zürich, Switzerland, ETH-Report Nr. 55, Sep. 1983.

Appendix:

(*
 LIDAS - Lilith Database System, Implementation for Lilith/PDP-11
 RDSREM: Relational Element Manager
 Version 2 of January 1983, Changed: March 1983
 LIDAS Research Project - Principal Investigator:
 C. A. Zehnder, Institut für Informatik, ETH-Zürich, CH - 8092 Zürich
 *)

DEFINITION MODULE RDSREM;
 (* Authors: Jürg Rebsamen, Manuel Reimer, Andreas Diener *)

FROM SYSTEM IMPORT ADDRESS;

IMPORT RDSDEM; (* database description manager *)

EXPORT QUALIFIED

(* variables *) systemKey,
 (* types *) Relation, Key, Mode,
 (* procedures *) Find, Obtain, Insert, Delete, Replace,
 BeginTransaction, CommitTransaction, OpenDB, CloseDB,
 InitRelation, InitKey,
 (* functions *) Card, Done,

(* The following types, procedures and functions should only be used by other system components (i.e. Query Evaluation Manager QEM and code generated by the Modula/R compiler). These procedures do not contain any plausibility tests and should be used with utmost care. *)

(* types *) Atr, AtrTypes, Comparison,
 (* procedures *) CreateRelation, CreateKey, CreateAttribute,
 CreateAttrInKey, DropRelation, ClearRelation, AbortTransaction, GetdbName,
 (* functions *) TestRelation, TestAttribute, Compare;

TYPE

Relation;
 Key;
 Mode = (first, last,
 next, nextequal,
 prior, priorequal,
 firstkey, keyornext,
 lastkey, keyorprior,
 current);

VAR systemKey : Key;

(* Relations are denoted by variables of type *Relation*. Access Paths (keys) are denoted by variables of type *Key*. An element of a relation *r*, with its corresponding relation element type, is referenced by *re*, which points to a variable (record). The type compatibility of element type and variable (record) type can't be checked. *)

PROCEDURE Find (r: Relation; re: ADDRESS; k: Key; m: Mode);

(* *Find* looks for an element of the relation *r*. The search follows the access path specified by *k*. No data is copied to the variable referenced by *re*. *Find* determines a new position within the relation *r*, i.e. a new actual relation element.

For positioning, using an absolute access mode (*firstkey*, *keyornext*, *lastkey*, *keyorprior*) the key value is taken from the appropriate field(s) of the variable referenced by the parameter *re*. The access mode *firstkey* (*lastkey*) delivers the first (last) element with the specified key value. The access mode *keyornext* (*keyorprior*) have the same effect, but the element with the next higher (lower) key value is returned whenever no element with the specified key value is found. Note, that the ordering within elements with the same key value is system defined, if a non-identifying key is used.

When using relative access modes (*next*, *nextequal*, *prior*, *priorequal*) the key value of the actual relation element is controlling the access. The relation element with the lowest (highest) key value for a given access path is retrieved by using the modes *first* (*last*). To find sequentially the next higher (lower) element, the access mode *next* (*prior*) must be used. All elements with a certain key value can be found by searching the first matching element (using the modes *firstkey* (*lastkey*)) and then repeatedly using the *nextequal* (*priorequal*) mode.

Each time after having changed the access path for a specific relation *r*, the actual relation element is lost and an absolute access is necessary for getting a new position within the relation *r*.

The access mode *current* is not applicable for *Find*.

For sequential processing of relations where the sequence of elements does not matter, the predefined "key variable" *systemKey* can be used. This key must neither be initialized, nor be used with the access modes other than *first* or *next*. *)

PROCEDURE Obtain (r: Relation; re: ADDRESS; k: Key; m: Mode);

(* *Obtain* copies a relation element to the variable referenced by the parameter *re*. The parameters *r*, *k*, *m* are described above (procedure *Find*). The access mode *current* is available as an additional access mode and fetches the relation element which was previously found by *Find*.)

PROCEDURE Insert (r: Relation; re: ADDRESS);

(* A new element, referenced by *re* is inserted into relation *r*. *Insert* guarantees the uniqueness of the identification key. Only for further processing along the access path *systemKey*, the inserted element becomes an actual element of the relation *r*. *)

PROCEDURE Delete (r: Relation; re: ADDRESS);

(* An element of the relation *r* which has identical identification key attribute value(s) as the variable referenced by *re*, is removed from *r*. After the deletion of a relation element in *r*, no actual element is defined for further processing, i.e. relative access modes can't be applied (see *Find*). *)

PROCEDURE Replace (r: Relation; re: ADDRESS);

(* The value of the variable referenced by *re* replaces an element of the relation *r* **with the same value for the identification key**. Note, that for further processing with *systemKey*, the element is not necessarily the actual element as before execution, as is after a successful *Replace* operation. *)

PROCEDURE Card (r: Relation): CARDINAL;

(* Returns the actual number of relation elements (cardinality) of *r*. *)

PROCEDURE InitRelation (relname: ARRAY OF CHAR; VAR r: Relation);

(* Before operating on a relation, it must be initialized by *InitRelation*. The relation with the name *relname* is referenced by *r* for subsequent RDSREM operations. *)

PROCEDURE InitKey (keyname: ARRAY OF CHAR; VAR k: Key);

(* In analogy to *InitRelation*, an access path with the name *keyname* is initialized by *InitKey*. *)

PROCEDURE BeginTransaction;

PROCEDURE CommitTransaction;

(* For the execution of any modifying operation on database relations (i.e. *Insert*, *Delete*, *Replace*) a transaction must be active, else the program is aborted. The scope of a transaction is determined by calls of *BeginTransaction* and *CommitTransaction*.

In case of hardware or software failures, the database will be recovered by the RDS to the state which was committed by the last transaction (i.e. aborted transactions are un-done). Therefore, the procedure *CommitTransaction* marks the end of a consistency-preserving statement sequence and guarantees that the actual content of the database becomes permanent and that subsequent failures can't touch committed transactions. *)

PROCEDURE OpenDB (dbName: ARRAY OF CHAR);

(* Before any operation on a database, *OpenDB* must be called. The name of the database must be specified by *dbName*, which must be compatible to *Files.FileName*. The file type (extension) is ignored. The database files used are "dbName[0..2];dbName[3..8].Dxx". If an empty string is supplied, the **default database files** is "DK:LIDAS.Dxx". The recovery files used are "RD:dbName[3..8].Rxx" and for temporary files: "dbName[0..2];dbName[3..8].Txx". The last two characters of the file type "xx" range from "01" to "86", depending on the amount of data handled by the database. *)

PROCEDURE CloseDB;

(* The last operation on the database should be *CloseDB*. It closes relations, keys and database files. A transaction must not be active. Do not operate on an already closed database. *)

PROCEDURE Done (): BOOLEAN;

(* *Done* allows to test whether the previous operation was successful (*Done() = TRUE*). Depending on the operation just executed, different error situations can occur. Fatal errors

cause an error message on system console and then the program is aborted by *HALT*. In the case of soft error, `Done() = FALSE` is delivered.

Operations on relations or access paths not previously initiated by means of *InitRelation* or *InitKey* respectively, lead to undefined pointer values and normally cause fatal run-time errors.

Detailed informations about (soft) failures of any operation can be gained by the application program, e.g. by means of additional *Find* or *Obtain* operations. *)

```
PROCEDURE CreateRelation (VAR r: Relation; elementLength: CARDINAL);
```

```
PROCEDURE CreateKey (r: Relation; VAR k: Key; ident: BOOLEAN;
  tree: BOOLEAN; keyLength: CARDINAL);
```

```
TYPE
```

```
Atr = RDSDEM.Atr;
```

```
AtrTypes = RDSDEM.AtrTypes;
```

```
PROCEDURE CreateAttribute (r: Relation; VAR a: Atr; offset: CARDINAL;
  type: AtrTypes; length: CARDINAL);
```

```
PROCEDURE CreateAttrInKey (a: Atr; k: Key);
```

(* The procedures *CreateAttribute* and *CreateAttrInKey* enable the query evaluation manager and the compiler to generate temporary relations with several keys. The procedures must be called in a meaningful sequence: the relation *r* must already be defined when creating attributes and keys; attributes and keys must already be defined when defining the key structure by calling the procedure *CreateAttrInKey*. Relation and key structure is implicitly defined by the order of the calls of *CreateAttribute* and *CreateAttrInKey* respectively.

The parameters of the procedures *CreateRelation*, *CreateKey*, *CreateAttribute* and *CreateAttrInKey* have the following meaning:

- VAR r: Relation; VAR k: Key; VAR a: Atr;
References to the new relation/key/attribute. *r* and *k* are subsequently used as parameters in procedures *Obtain*, *Insert* etc., whereas *a* is exclusively needed by *CreateAttribute* and *CreateAttrInKey*.
- elementLength: length of the element of the relation *r* (in words).
- keyLength: length of the total of all components of the key *k* (in words).
- ident: TRUE if the new key *k* is the identification key, FALSE otherwise.
- tree: TRUE if the new key *k* is to be supported by a tree, FALSE otherwise.
- offset: Offset of attribute in relation elements, i.e. within users record type (in words). The first attribute of a relation has therefore offset = 0.
- type: The type of this parameter is defined as follows: `AtrTypes = (atrINT, atrCARD, atrBOOL, atrCHAR, atrREAL, atrSTRING)`;
Select `atrCARD`, if you like enumerations as attribute types in your relation. For subranges select the corresponding base type.
- length: Length of attribute in relation elements (in bytes).

Temporary relations can be operated also outside of a transaction. *)

```
PROCEDURE DropRelation (r: Relation);
```

(* This procedure is used by Modula/R programs at the end of a scope of a local (temporary) relation. All data pages are returned and the entire relation description disappears. Do not drop permanent relations ! *)

PROCEDURE ClearRelation (r: Relation);

(* ClearRelation is used by the query evaluation manager QEM. An existing relation (local relations, auxiliary relations) is reinitiated, i.e. all elements disappear but the description entries to the relation remain available for subsequent use. *)

PROCEDURE TestRelation (r: Relation; elementLength: CARDINAL);

PROCEDURE TestAttribute (r: Relation; offset: CARDINAL;

type: AtrTypes; length: CARDINAL; keyAtr: CARDINAL);

(* The procedures TestRelation and TestAttribute may be called after a previous call of InitRelation and InitKey of the identification key. The Modula/R compiler inserts calls of these test procedures in order to guarantee at least the type compatibility between the database relations and the relation variables of the same name within the program. The parameter *keyAtr* describes the position of the attribute within the identification key of the relation *r*: *keyAtr* = 0 means that the attribute is not part of the identification key. Implementation restriction: Whenever more than one identification key to one relation is defined, the first one is regarded to be the Modula/R RelationKey. The precise meaning of the remaining parameters can be examined in the definition module of RDSDEM. These procedures return, if no type incompatibility is detected, otherwise the program is aborted with RDS Error 305. *)

PROCEDURE AbortTransaction;

(* In case of abnormal situations the system programmer can abort a running transaction without terminating the entire program. The effects are summarized in the following:

- Previous modifications (i.e. updates which have been performed by the aborted transaction) on permanent (database) relations are made undone.
- Temporary relations and program variables are not altered by AbortTransaction. The programmer himself is responsible for correcting these values and perhaps making undone several side-effects. *)

TYPE Comparison = (less, equal, greater);

PROCEDURE Compare (val1, val2: ADDRESS; type: AtrTypes; length: CARDINAL):

Comparison;

(* Compares two values, *val1* and *val2*, according to their type described by type. Values of any type (belonging to AtrTypes) may be compared, but both must be of the same type. The address indicated by *val1* and *val2* must be the address of the *values*, not of the *relation element*. *)

PROCEDURE GetdbName (VAR dbName: ARRAY OF CHAR);

(* returns the name of the database. *)

END RDSREM.