

Title:

S O D A

2nd edition

 **REGNECENTRALEN**

RC SYSTEM LIBRARY: FALKONERALLE 1 DK-2000 COPENHAGEN F

RCSL No: 21- Vo54 (pp 198)

Edition: Oktober 1979

Author: Paul Lindgreen,
Edith Rosenberg

Keywords: RC4000, RC8000, SYSTEM80, database, DBMS, Subschema, DB access, inter-record structures, logical data description, record set, subscripted set, external declarations, standardization, data independence, ALGOL, DUET, DATABASE80.

Abstract: This manual describes the SODA system. SODA comprises a database management system (DBMS), a formal language for local data description (subschema/external schema) and a compiler to process such a description. The DBMS provides convenient and powerful DB access possibilities based on record sets, which is a generalized version of the set type known from CODASYL like systems.
English edition.

Preface

The SODA system is a serious attempt to raise the level on which application programmers must work when the program operates on databases. Through intensive use of specifications in a tabellized manner outside the program, by application of declarative compiler technique, and by means of a certain degree of standardization, the programmer can now concentrate on the logic and the tasks of the application. The tedious and often complicated set up of parameters to the available file systems is removed from the application program and taken over by the SODA DBMS. This manual introduces the concepts behind SODA and describes all the features, rules and principles that must be known in order to utilize SODA.

The system was designed by the present author together with Edith Rosenberg. The programming was done by the same two persons together with Isabella Carstensen.

Paul Lindgreen



CONTENTS

1. Introduction	1
2. System survey and basic concepts	7
2.1 The main components of SODA	8
2.2 The SODA record accessing scheme	13
2.3 The SODA field accessing scheme	27
3. The SODA Local Data Description	31
3.1 Declaration of SODA variables	33
3.1.1 The variable table	33
3.1.1.1 Variable declaration without numbers	40
3.1.2 The value spectra table	41
3.1.3 The norm value table	43
3.2 Declaration of record sets	45
3.2.1 The set declaration head	46
3.2.2 Set restrictions	49
3.2.3 The usage specification	52
3.2.4 The log specification	53
3.2.5 The ident specification	54
3.2.6 The mother specification	57
3.2.7 The daughter specification	63
3.2.8 The field specification	66
3.3 Automatical declaration of record sets	77
3.3.1 Record output	77
3.3.2 Implicit variable declarations	80
3.3.3 Record input	85
3.4 Connection to the Data Base Description	87

4.	The SODA LD compiler	89
4.1	Activation of the compiler	92
4.2	Listing and log	104
4.3	Reaction on errors	112
5.	The SODA DBMS	115
5.1	GET	119
5.2	NEXT	121
5.3	LOOKUP	123
5.4	PUT	124
5.5	CREATE	127
5.6	DELETE	129
5.7	NEWSET	131
6.	How to include the SODA DBMS in a program	133
6.1	ALGOL block structure and SODA program texts	134
6.2	Initiating the SODA DBMS	137
6.3	The log mechanism	141
6.4	The DBMS error mechanism	144
6.5	Reserved ALGOL identifiers	148
Appendix A.	Formal description of the SODA LD syntax	150
Appendix B.	Error messages from the SODA LD compiler	165
Appendix C.	References	184
Alphabetical index		185

1. Introduction

SODA This manual describes the so-called SODA system for RC 4000/8000. The description is intended for system analysts, programmers, and datamatic consultants.

SODA, which is an acronym for Set Oriented Database Access, is the name of a datamatic tool to be used for flexible and convenient access of a common database (DB) from individual application-programs.

Components

Access from a given program is performed by a DBMS (database management system) which is incorporated as a part of the program. The DBMS is governed by information derived from a program-specific local data description, which again refers to those parts of a common DB description that are relevant to the program *). SODA consists of the DBMS, a local data description language called SODA-LD, and a compiler for this language.

The SODA system realizes two important access principles:

Record access via sets

As the name indicates the DB access is based on the concept a record set. A record set in SODA is in many respects similar to a set type in the CODASYL proposal (ref.7), but in SODA it is considered in a more general and simple form resulting in a much easier access scheme.

*) Readers familiar with CODASYL terminology will observe that we use the term 'local data description' for 'subschema' and 'DB description' for 'schema'.

In the LD description any number of record sets can be declared, each one defined as a subset of the records belonging to a logical file in the DB. Access to the records is performed entirely with reference to a record set, whereby it becomes possible to have one record 'at hand' at the same time for each set, regardless of how the logical files are associated with physical files. The definition of the sets can be specified as a list of relevant record types and furthermore by a general restriction expression in values of record fields and program variables.

Field access
via variables

The second access principle realized in SODA is that references to record fields for use or for updating take place via ordinary variables in the employed programming language. In the LD description a two way mapping between fields and variables is specified. In the DBMS read or write operations values are transferred in accordance with this mapping. In this way the programmer does not need to know or refer to buffers or zone records.

Data
independence

The implementation of these principles provides a solid base for a high degree of data independence, i.e. that each application program is less sensitive to logical extensions or physical reorganizations of the DB caused, for example, by new requirements to other application programs.

ALGOL and
DUET

The present implementation of SODA permits the programmer to employ ALGOL 6 as well as DUET (see ref. 5) as programming language. In nearly all respects the DBMS will operate in the same way whatever language is used.

Advantages
of SODA

SODA is designed and implemented with the primary intention to facilitate the implementation and maintenance of application programs. The main advantages obtained through the use of SODA are:

- 'logical view' of the relevant part of the DB with record grouping suited to the task of the application program.
- no complicated parameter lists for the accessing operations.
- very extensive checking against all kinds of formal errors both in the LD description as well as dynamically. This will minimize the effort necessary for debugging and improve the integrity of the DB.
- improved and well structured documentation of use of the DB.
- more straightforward logic and fewer statements in the application program.
- support of repeating groups, possibly resulting in faster or simpler application programs.

Drawbacks of
SODA

Owing primarily to limitations in the underlying file systems (ref. 1 and 4), SODA does not permit concurrent access to the same files from more than one program. Furthermore, because the DB description (ref. 2) does not support any data protection no precautions have been taken in the implementation of SODA to remedy this.

Although these missing features could seem to be serious disadvantages in certain cases, this should not prevent the user from employing SODA. In most cases the use of SODA will lead to much simpler and easy-to-maintain systems than if DB access is based directly on the file systems, which, anyway, do not support concurrent access.

Guide for the
new reader

If you receive this manual for the first time and have no, or only a limited, knowledge of SODA, you are advised to read it in the following way:

- 1: Read section 2 in lexicographical order, but ignore reference to more detailed descriptions in other sections. Only exceptions are references to figures, which may help you. Readers are assumed to be familiar with the Database 80 language (ref. 2). A general knowledge of the CF system (ref. 1) is probably useful but is not absolutely necessary.

- 2: If you have time enough then read the paper in ref. 3, but be aware of a slight change in the terminology in some cases.
- 3: You should now have a reasonable general overview of SODA. If you are lucky to have access to persons who have used SODA (they do exist and some have learned it the hard way without this manual) then discuss any mysterious or unclear points with them, but do not get stuck in details. You still have much to learn.
- 4: Select a simple application problem. If you do not have one, invent one. It must be one referring to a DB where a DB description is available. Now, read slowly, possibly iteratively, through section 3 and follow all references and local hints as long as you feel the jumps in reading will help you to understand. Parallel to the reading try to write down selected parts of the LD description relevant to your application problem. If you feel you cannot do it, reduce the complexity of your problem. It can be done !
- 5: Now read section 5 guided by a parallel programming of your application program at some reasonable level of abstraction (skip details irrelevant to the logic of DB accesses).

- 6: Finish your LD description, at least to the point where it is formally correct (or you think it is). Then read section 4 and try to run a compilation of your LD description. Correct all errors and continue until your description is formally correct.
- 7: Read section 6 fast, just to get an impression of it. If you shall use DUET as programming language you may possibly skip this section.

Now you should be prepared for the troubles of really using the manual. Good luck !

2. System survey and basic concepts

This section gives a broad survey of SODA and introduces most of the notions and concepts on which a thorough understanding of the system is based. The text is organized so that the reader should be able to follow it as it appears. However, since the remaining sections assume the reader to be familiar with all the concepts presented here, it may later on be necessary to consult certain parts in order to be sure of a correct understanding.

The section is divided into three parts:

First, the principal components of the system are presented and the interface to other systems or tools is illustrated. Then, in two other parts, the two main access principles are described - how records are selected and transferred to and from the DB and how the individual fields of the records are manipulated from the application programs.

2.1 The main components of SODA

The SODA
LD description

Each application program intending to use the facilities of SODA must be associated with a formal description called the SODA LD description. It specifies those elements of the DB which are relevant to the application program. Normally an application program will have its own LD description, but it is possible that one description is common to several programs.

The LD description can be regarded as an external set of declarations to the application program. In the selection of relevant parts of the DB, the LD description refers to entries in a common DB description expressed in the Database80 language (see ref. 2).

The LD description contains declarations of two kinds of entities - SODA variables and record sets.

SODA variables

A SODA variable is a variable of one of the types permitted in the programming language employed. The head of the LD description contains a reference to this language so that the SODA LD compiler can check that only variables of legal types are declared. Apart from this, the declarations are independent of the programming language - also concerning the syntax. A SODA variable is primarily used for access of record fields as described in 2.3, but aside from this they can be used freely in the program. In case

of ALGOL, SODA variables will appear exactly as other variables declared in the ordinary way; in the case of DUET, it is the only way variables can be declared.

Record sets

For a number of reasons, access of records with a direct reference from the program to the files in which they are stored is not supported by SODA. The intention is to allow the designer and programmer, as far as possible, to think in logical data structures rather than in physical ones. Therefore, in the LD description, the user must declare one or more record sets as groups of relevant records which are candidates for access. The record accessing scheme is explained in detail in section 2.2.

THE SODA LD language

The LD description is expressed in a formal language called SODA LD. This language is explained and described in detail in section 3 while a formal syntax description is available in appendix A.

The SODA LD compiler

When the LD description is finished - or at least assumed to be formally correct - it must be checked and processed by the SODA LD compiler. It reads the LD description and the referenced parts of the DB description (from its internal form in the DB description file) and checks the specified information for completeness and formal consistency. A format-edited listing of the LD

description is produced together with possible error messages. If the description is formally correct, the compiler will produce a binary file - the SODA LD file - containing the information necessary for the DBMS to operate properly (see fig. 2.1). The compiler may also produce a text file with ALGOL declarations of variables to be incorporated in the application program (cf. section 3.1.1). The function of the compiler and the rules for its activation are described in detail in section 4.

The SODA
DBMS

The DBMS is an ALGOL text containing declarations of:

- procedures corresponding to the DBMS operations for direct or sequential reading and updating of records and for creation or deletion of records.
- procedures for initialization, testing etc.
- variables and tables to be used by the procedures.

If DUET is chosen as programming language, the user should note that these declarations are a part of the DUET system programs. The initialization of SODA, the DBMS operations, and the dynamic error handling are fully integrated in DUET as described in ref. 5. However, when the user employs ALGOL as programming language the above-mentioned text must be incorporated in the application program itself. This is described in section 6 of this manual.

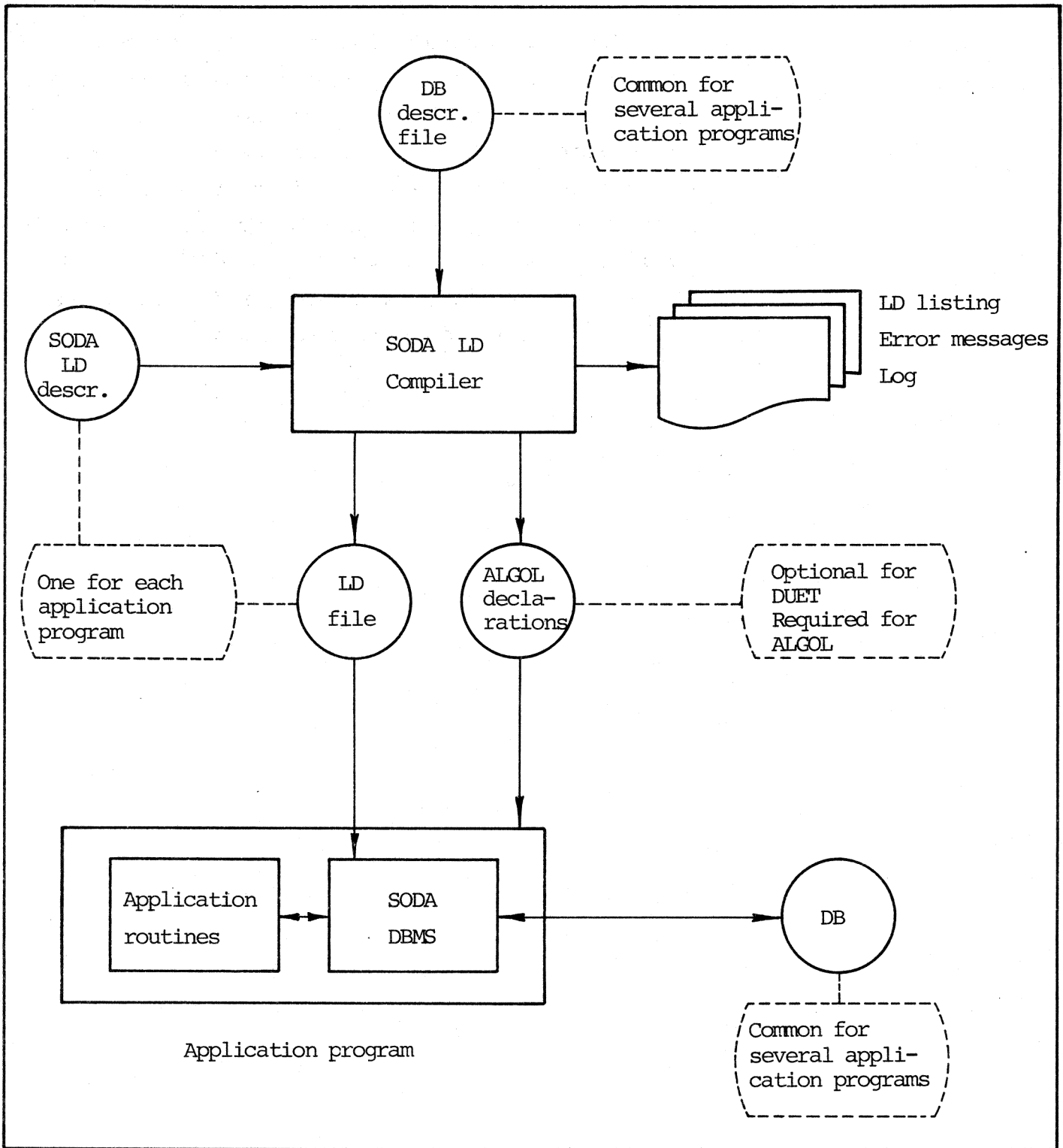


Fig. 2.1 General survey of SODA component relations

The DBMS will operate in accordance with the activation of the various procedures from the application program, but governed by the information stated in the LD description and represented in the SODA LD file. The initialization procedures will read the data stored in this file into the internal tables of the DBMS and the accessing procedures will then refer to these tables for the necessary information.

2.2 The SODA record accessing scheme

SODA enables the user to regard the database as a collection of records of various types organized in record sets.

Record sets

A record set (RS) is a collection of records which have something in common of interest to the application program. Normally, it reflects an entity class in the user system about which information is represented in the database.

Every record type that a given program wishes to access must be specified as a member of at least one record set declared in the corresponding LD description, but the same record type may be a member of more than one set.

Different application programs are permitted to have their own individual views of the database, both as regard to the subset of interest and the grouping of record types into record sets.

Physical dependence

However, the grouping may not violate the physical structure of the database as defined in the DB description. This is because the SODA operations (see 5) are implemented by means of the available file systems which draw heavily on the organization of records in different types of physical files (CF master, CF list, and BS sequential (ref 1 and 4)).

Set types

Accordingly, SODA will distinguish between different set types primarily corresponding to the above mentioned physical file types. In most respects, sets of different types are treated the same way - at least from the user's point of view - but there are some differences. In the description we shall refer to set types denoted like this:

Settype M: Sets associated with CF master files.

Settype B: Sets associated with BS files.

Settype L: Sets associated with CF list files.

Record access
operations

Access of records in the database is performed from the application program with reference to a declared set by activation of one of the SODA DBMS operations listed below.

GET provides a directly accessible record belonging to a set from the DB, according to specified ident field values.

NEXT provides the next record in the sequence of a set from the DB.

LOOKUP verifies that a directly accessible record is present in the DB as the member of a set.

CREATE generates a new record to be inserted in the DB.

PUT returns a record read by GET/NEXT to the DB or inserts one generated by CREATE in the DB.

DELETE removes a record belonging to a set from the DB.

The use of any of these operations must be announced to SODA in the set declaration and must furthermore follow a set of rules dependent on the set type, the dynamic sequence of activation of the individual operations, and on the DB itself. The set of rules cannot be described in a single scheme nor can a straightforward classification of the rules be made. Because of all these logical dependencies, in the design of SODA much attention was given to obtaining uniform, general, and simple functions. However, constraints imposed by the file systems and the DB description as well as attempts to cover most user applications in a reasonable way have caused some ad hoc rules and certain limitations in the intended generality.

Direct read
access

via ident
fields

The fetching of an individual record, regardless of any previous access operation on a set, is performed by the operation GET. The information necessary to identify the record in question - the so-called key values - must be supplied from the application program in one or more variables specified in the LD description as a part of the set declaration (see 3.2.5). For sets of type M it is necessary to provide a (user-specified) value for each ident field in the

via recno

records as defined in the DB description. For the other two set types, the records are identified by the so-called recno value. This is a system-generated value which is available to the application program originally when the record is inserted in the DB by a PUT operation after a CREATE and later on after every NEXT operation that provides the record. *)

Sequential
read access

For all set types it is possible to perform a sequential scan of the records belonging to the set, regardless of possible direct access operations on the same set. SODA maintains the necessary information about which record was last accessed by a NEXT operation and which one by a GET operation. The NEXT operation will always provide the next record in the set relative to the one last accessed by NEXT. The sequence of records in the set is dependent on the set type in the following manner:

M: Increasing values of ident fields

B: Physical location in secondary storage

L: Position in the chain used for access

*) For set type L the recno value is identical with the record number provided by the CF system. For set type B it is defined as follows:

segment_no shift 8 add baseword_adr

- Scan initialization (NEWSET) A scan must be initiated by activation of the operation NEWSET. Then NEXT can be activated repeatedly until it, as result, announces that the last record of the set was provided. At this time, or at any previous time, NEWSET can be activated to indicate the start of a new scan.
- Set sequential status The sequential access of records in a set is thus characterized by two values of what is called set sequential status.
- closed: Before first NEWSET or after the NEXT of a scan that announces 'no more records'
 - open: After NEWSET, but before the above mentioned last NEXT of a scan.
- Set sequential position NEWSET will locate a pointer - the set sequential position - to a certain location in the sequence of records in the set. Normally this position will be just in front of the first record, but for settype M and B it is possible to locate the position to any record belonging to the set. The location is defined for NEWSET in the same way as a record is identified for GET (see 3.2.5).
- The set sequential position will be modified whenever NEXT is activated, but it will not be affected by a possible GET in between.
- Record updating When a record has been read by GET or NEXT, the application program can use or modify the record fields according to specifications in the set

declaration as explained in section 2.3 and 3.2.8. If the values of one or more fields are changed, the record must be returned to the DB to replace the previous occurrence of it. This is done by activation of the operation PUT.

Record creation

The insertion of new records in the DB is performed via a set in full accordance with the record definition in the DB description. The insertion takes place in two steps: First, by activation of CREATE, a record of a type defined for the set is generated and provided for the application program. The various fields are initialized to standard values in accordance with specifications in the LD description. Second, by activation of PUT, the generated record, with fields possibly modified by the program, will be inserted in the DB. The location where the record is inserted is dependent on the set type in the following manner:

Insert location

- M: The location of the record is defined by values of the ident fields communicated in the same way as for direct access. Set sequential position is not affected.
- B: The record is inserted just after the last record in the physical file. Set sequential position is not affected.
- L: The record is inserted just prior to set sequential position.

Record deletion

In order to remove a record from the DB it must first have been read from the DB by GET or NEXT. Then an activation of the operation DELETE will remove the record. For records declared in the DB description to be mother records (see ref. 2 or the entry "Subscripted sets" below), some further rules must be obeyed (see 3.2.7 and 5.6).

Current record

For each set, one and just one record can be available at a time for the application program to operate on. This record is called the current record of the set.

A current record is obtained either by a read operation (GET/NEXT) from the DB or by a CREATE operation. It is released again by a subsequent PUT operation or, if the user wishes to remove it, by a DELETE operation. Only when a current record is available in a set is it possible for the application program to operate on it - that is, use its field values or modify them.

Record status

Accordingly SODA keeps track of the current record situation for each set. From the users point of view this is represented in the so-called record status which in many respects defines the legality and the function of the DBMS operations. The possible values of record status for a set are:

- empty : No current record in the set
- DB currec : Current record read from DB
- new currec : Current record established by CREATE

- Record availability The same record type (in fact even the same record) may be available as current record in more than one set at the same time. For certain applications this will result in a much simpler program logic with less need for intermediate storage. On the other hand, if this feature is misused, SODA may be forced to perform many, otherwise unnecessary, physical file accesses. The descriptions in sections 2.3 and 5 should give the user the necessary information to decide which is the optimal solution in a given situation.
- Set declarations The record sets of relevance to the application program must be declared in the LD description. A set declaration defines an identification of the set, it refers to a logical file declared in the DB description *) (see ref. 2), and specifies the record membership of the set (see fig. 3.9). Furthermore it defines which fields of the records the application program wishes to use or update as explained in section 2.3.
- Record membership Sets may be declared to comprise all the records of a logical file or just a subset of it. Subsets may be specified either as a list of selected member record types or by means of a logical expression (possibly one for each record type) or both. The selection expressions may be composed of relations separated by the logical operators 'and' or 'or' and the relations may refer to

*) In the Danish text of ref. 2 the term 'register' is used for 'logical file'.

fields, constants, or SODA variables (see fig. 3.10) . The logical expression defines membership as those records of the relevant types for which the expression is true at the time of access.

The SODA view
of the
information
structure

The representation in the DB of the information structure in the user system will normally be so that different entities are represented as different records. The various sets of characteristic properties of the entities are reflected in the DB in corresponding record types. Each record type is defined in the DB description by a set of fields representing attribute as well as relational properties. In the DB description the record types are partitioned into logical files, but SODA enables a further organization with a possible overlapping grouping of record types in record sets reflecting different application oriented properties.

When the information structure comprises entity relations of the one-to-many kind this is represented as inter record structures where a set of so-called daughter records by a chaining technique are linked sequentially to each other and to a single mother record. Physically the daughter records must be stored in a CF list file (see ref. 1) and normal access will require a previous access of the mother record. In the mother record the one-to-many relational property is represented in a so-called d-ref field *).

*) In the Database80 language (see ref. 2) d-ref fields are declared as type 'list'.

Set kinds

In SODA these two different organizations of records are reflected in a distinction between two kinds of record sets - singular sets and (mother) subscripted sets. This distinction is in some way similar to that known from programming where the programmer must consider both simple variables and arrays.

Singular sets

A singular set in SODA is a set where the records are accessed and set membership defined by a reference to the set declaration only. The records of a singular set can be accessed directly or sequentially, in general as explained above. The only exception is for singular sets of set type L. Here only GET and PUT can be employed and only if the recno identification is known.

Subscripted sets

A mother subscripted record set in SODA is a set where the records are organized in separate groups, so that each group is associated uniquely with a single record (the mother record) belonging to another set - the mother set.

Access of the records in the mother subscripted set - the daughter records - can take place only within one group at a time. Whenever the program wishes to access records belonging to another group that group must be selected separately before the access can take place (see fig. 2.2).

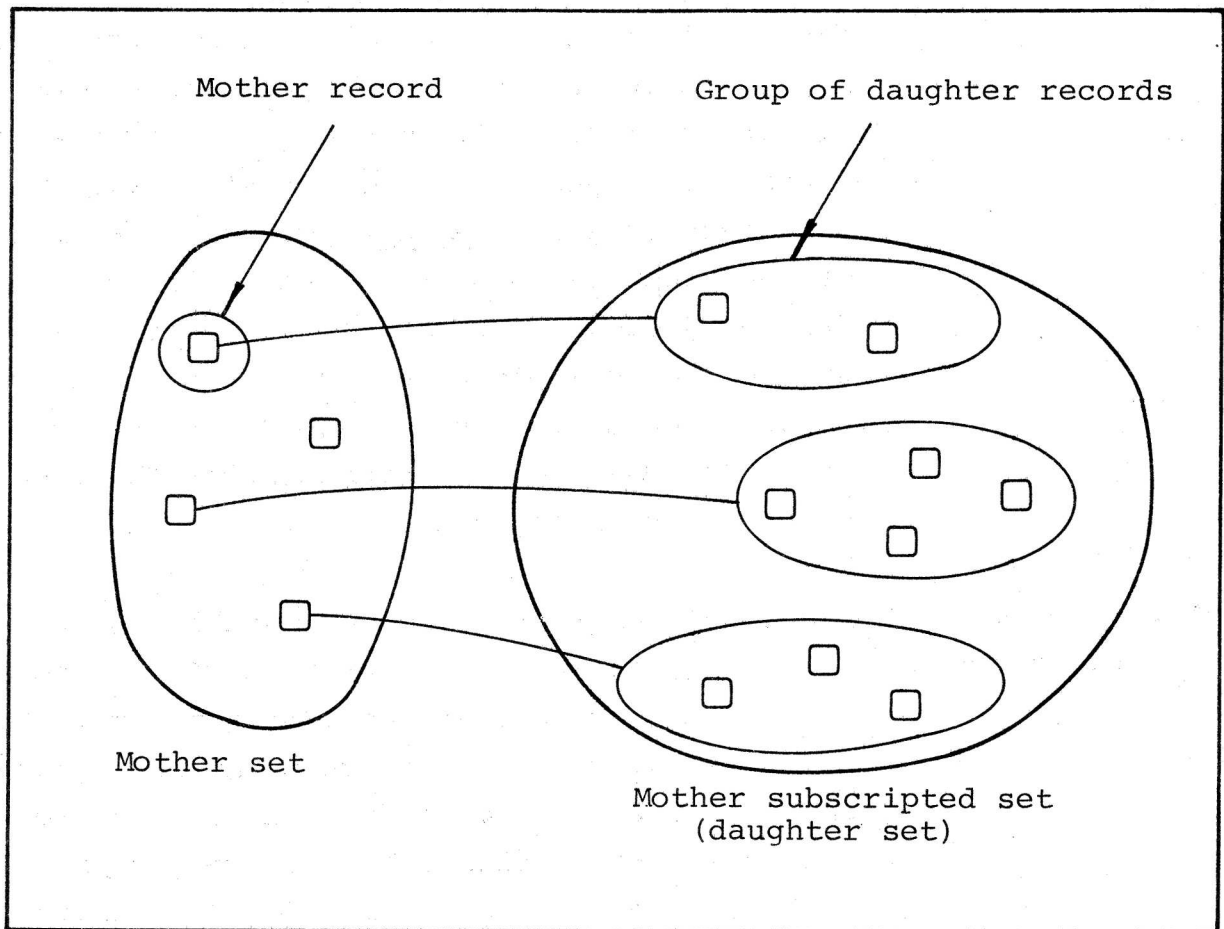


Fig. 2.2 Principle of mother subscription

Array analogy

The analogy to arrays now becomes clear: An array is a collection of elements each representing some value. In most programming languages the elements are identified by a number called the subscript or index. In SODA an element of the 'array' is one of the groups of daughter records and the identifying subscript is the mother record.

Subscription

The selection of a group is performed in two steps. First, the appropriate mother record must be accessed by a read operation whereby it becomes the current record of the mother set. Second, the operation NEWSSET must be activated with reference to the daughter set. The records of the group can now be accessed, created and deleted just as if it was a singular set. The reason why the subscription requires two steps is to enable access of (other) records in the mother set, regardless of the treatment of the records belonging to a selected group.

Multi mother linking

In the DB description it is possible to declare inter-record structures so that a given record is linked to more than one mother record. This reflects information structures where a given entity class is many-to-one related to more than one other entity class. The user may regard this so that a set of (daughter) records is organized in groups in several independent ways. Each set of groups is associated with its own set of mother records as shown in fig. 2.3 and the daughter records can then be accessed from whichever mother set is convenient.

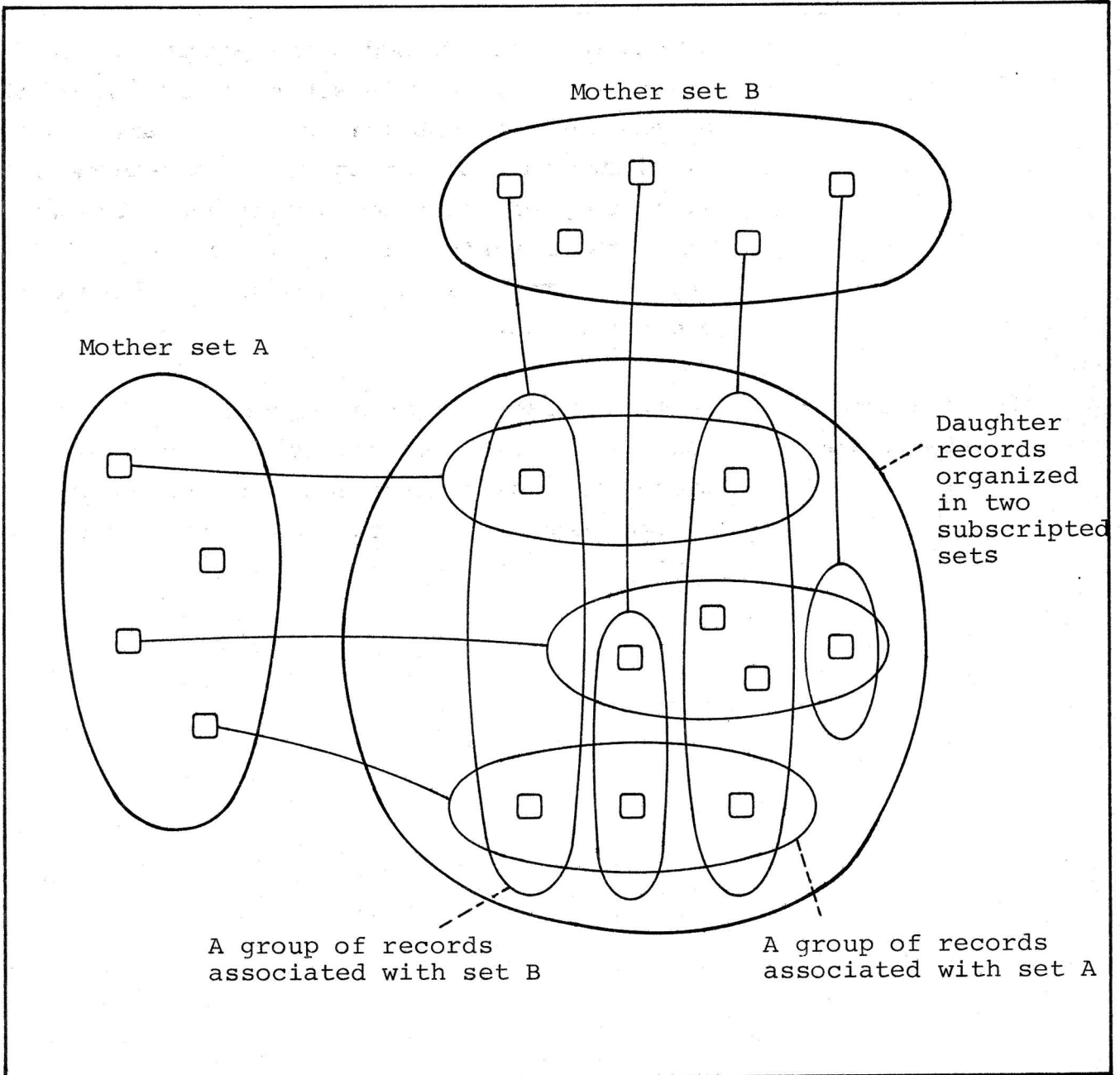


Fig. 2.3 Independent subscription in two dimensions

Array analogy
again

The analogy to arrays is still valid. As probably well known, arrays in most programming languages can be declared to have more than one dimension, each one with its own set of subscript values. The program may then choose to operate on the elements in a row or it may choose to operate on them

by column, each method corresponding to the selection of a certain subscript in one of the dimensions. In SODA this holds for any number of dimensions, but in the present implementation it is only possible to operate on 'sub-arrays' with one dimension fixed. This corresponds to the rule that a subscripted set can have only one mother set.

Declaration of
subscripted
sets

In the LD description a subscripted set is declared in the same way as a singular set except for the declaration head, which contains a reference to the mother set (see s8 in fig. 3.9).

2.3 The SODA field accessing scheme

Zone records

From an ALGOL program it is possible, via field addressing of zone records, to refer to attribute fields of records which are accessed by procedures of the file systems (see ref. 8). A DUET program cannot refer to zone records, since zones are not defined in the language, and from an ALGOL program the zone reference method would be violated by

Not in SODA

the SODA record accessing scheme: During access each physical file is bound to exactly one zone and only one record is available as the zone record for each file. SODA, however, maintains one current record for each set and permits several sets to be associated with the same physical file.

Access via SODA variables

For this and a number of other reasons (see ref. 3), SODA provides access to record fields via the declared SODA variables. When a record is read from the DB, values of the relevant fields are transferred to a set of variables and before a current record is returned to the DB, values from another set of variables are transferred to those record fields which the program is to update. The two sets of variables may be identical, partly overlapping, or quite distinct, just as it is appropriate to the application. The field access principle is shown in fig. 2.

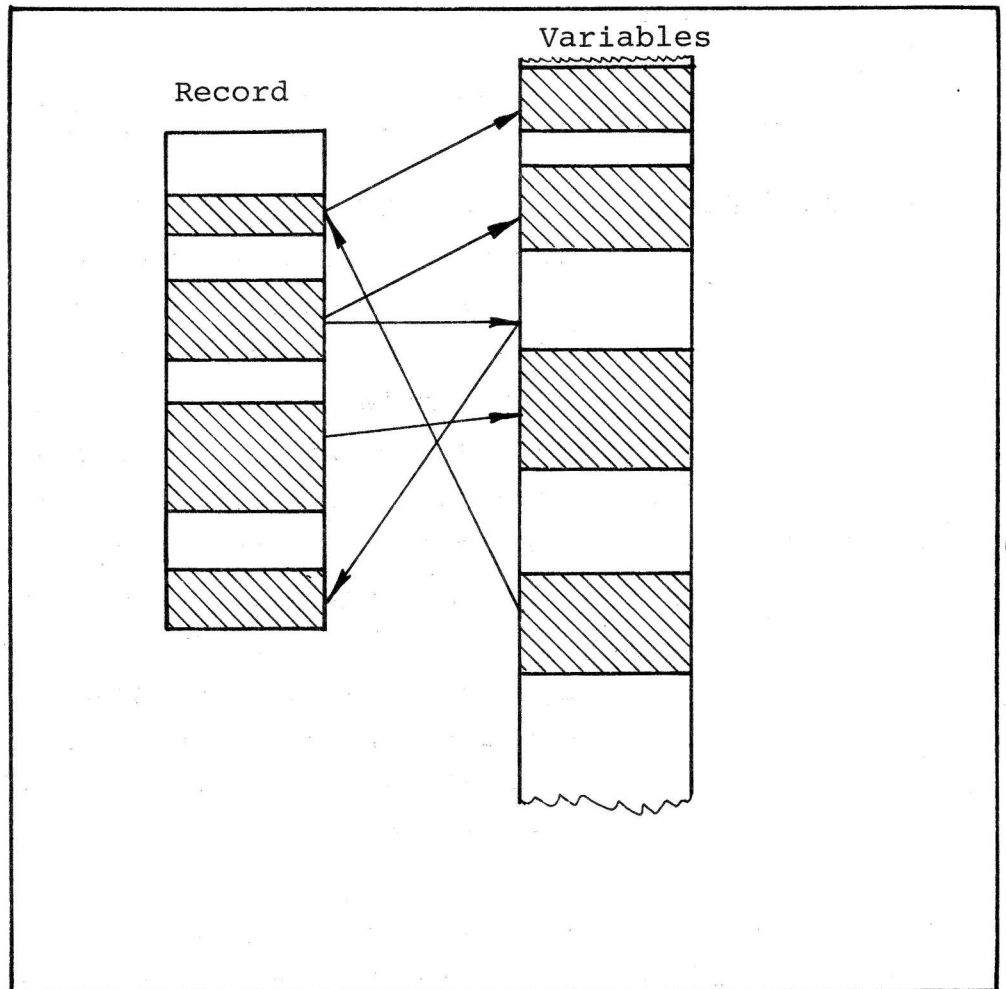


Fig.2.3. Principle of transfers between fields and variables

Shorter
programs

Especially in administrative data processing the transfer of values from fields in one record to fields in other records is a very common operation requiring numerous statements of the type $a:=b$ scattered throughout the program. Since the relevant field values in accordance with the chosen principle in SODA must be moved anyway, it seems obvious to associate the same variable both with the field in the yielding record and the field in the receiving record, thereby saving an assignment in the program text as well as in the execution.

In many cases this will result in a considerable reduction of program length. SODA, on the other hand, may make the reading and understanding of a program more difficult, because the detailed information is represented in two documents instead of one.

The field
specification

The mapping between record fields and variables is specified for each set in the LD description. In a section of the set declaration called the field specification any number of field/variable associations can be specified, each defining a transfer from a field to a variable, a transfer from a variable to a field, or a transfer in both directions. The direction of transfer is specified by a symbolic arrow-like operator between the references to field and variable (see fig. 3.23). Transfer can be specified for all kinds of fields and variables - simple elements as well as whole arrays. Constants may be transferred to fields. Transfers to and from elements of arrays with either a constant or a variable subscript, are also possible. Elements of repeating groups in records are in most respects treated as arrays with variable length.

CREATE
transfers

When a current record is established by CREATE, it is possible to have standard values transferred to variables which are associated with fields in the created record. (Fields not associated with variables will be assigned zero or empty anyway). A PUT following a CREATE will cause values

to be transferred to the fields from a set of variables which may be more or less different from the one used for PUT after a read operation. These destructions are also indicated by the symbolic transfer operator.

Ident
specifications

A special case of field/variable association must be considered for values of ident fields in records which are to be accessed directly. For sets where records are accessed by GET (and for settype M, if CREATE is employed, the user must specify which values are to be used as keys for the record. This is done in a separate section of the set declaration called the ident specification. Here, for settype M, each ident field is associated with a variable or a constant. For the other settypes, the reserved word 'recno' is associated with a variable.

When GET (or, for settype M, CREATE) is activated, the record to be the current one will be identified by the values specified in the ident specification of the set.

Independent of the ident specification the values of ident fields can always be transferred to variables according to associations in the field specification. Transfers to ident fields are only performed for settype M at the time of creation according to the ident specification.

3. The SODA Local Data Description

As mentioned in section 2 the user must declare the record sets needed by the program for access to the database and the variables through which the fields of the records are accessed. The declarations are specified in the SODA LD description by means of a formal language described in this section. Section 4 explains how the LD description by means of a compilation is prepared for use by the program and the DBMS. The complete syntax description of the SODA LD language is given in appendix A.

The overall structure of an LD description is as shown in figure 3.1

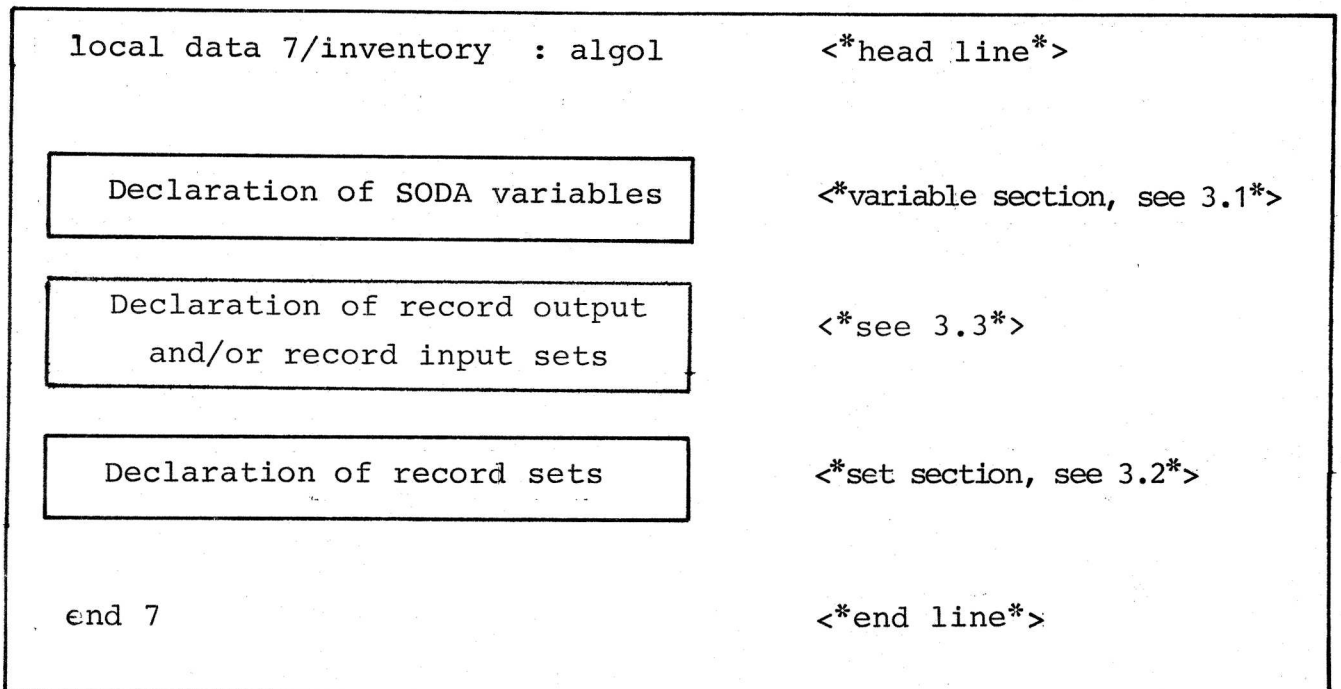


Fig. 3.1. Overall structure of a SODA LD description

Every line of the description may be terminated by a comment enclosed in the short-comment brackets <* *>. Between the various sections and subsections of the description any number of empty lines or lines containing a comment only are permitted.

The head line contains the reserved word 'local data' followed by a user specified identification and a specification of the programming language used for the application program. At present the two languages ALGOL and DUET for RC 4000/8000 can be used.

The identification consists of a number and a name separated by a /. The name should be an identifier with no more than 17 characters. The identification together with a version number will identify the compiled LD description in the LD file. It will secure that a program at runtime is executed with the correct LD file present to govern the SODA operations (see 5) and that a DUET program is translated based on the correct LD description (see ref. 5).

The end line contains the reserved word 'end' followed by the same identification number as in the head line.

3.1 Declaration of SODA variables

The variables used in the application program for access of record fields must be declared in the variable section of the LD description. It is composed of three subsections each headed by a separate line with an appropriate reserved word as shown in figure 3.2

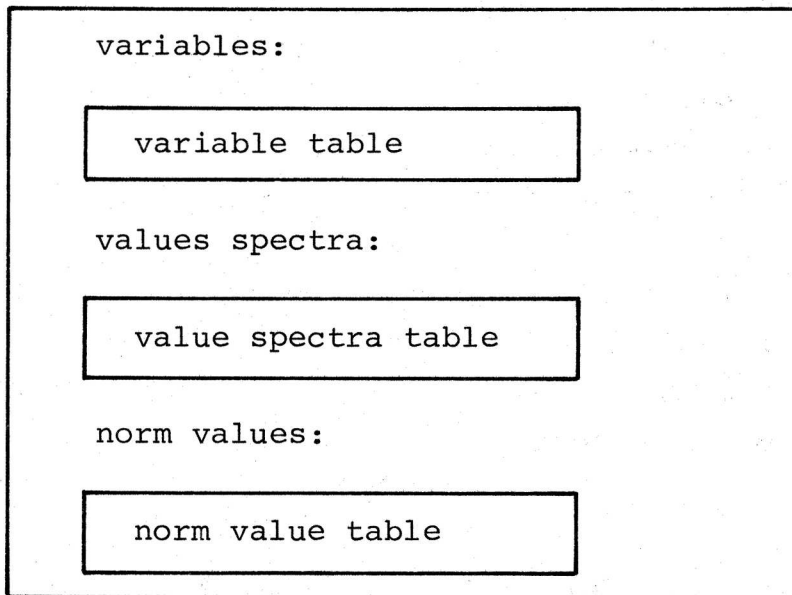


Fig. 3.2 Variable section

3.1.1 The variable table

The variable table consists of a number of lines each one declaring a single variable. The declaration can be specified in various ways most of which are illustrated in the example in figure 3.3.

```

variables:
  v1  : customer_number      : word
  v2  : balance              : long.2   w6   n2
  v4  : customer_ident      : text.29(5)   n5
  v5  : stock_on_hand       : real (2) w17
  v6* :                      : date
  v7  : order_spec          : bits.8
  v10 : order_line_key      : recno
  v11*: soda_ok             : result.soda
  v12 : vendor_ident        : = vendor_name

```

Fig. 3.3 A piece of a variable table

An ordinary variable declaration is composed of three main parts: an identification, a type specification, and possible value spec and norm value references as shown for the variables v1 - v11. An alternative way is to declare a variable by reference to a field in the DB description as shown for v12.

The identification of a variable consists of a variable number composed by a 'v' followed by an integer, and of a variable name which is an ordinary identifier formed by letters, digits, and the character '_' as a connector. The connector is significant in the recognition of identifiers in SODA LD. In this way the two identifiers soda1 and soda_1 are different. The variable name may be left out in which case the variable number is regarded as the name. Both variable numbers and names must be unique in the LD description. It is advisable (but not required) to let the declarations appear in ascending order of the variable number. Single numbers or intervals of numbers may be left out.

The type specification defines the type of the variable and whether the variable is simple or an array. A variable may be declared as a so called explicit variable by indicating one of the following types:

word: 24 bits. Equivalent to the ALGOL 'integer' type

long: 48 bits. Equivalent to the ALGOL 'long' type.

The types 'word' and 'long' may be followed by a decimal point and an integer indicating an implied number of decimals. This is fully utilized by the DUET system, but will have no influence when ALGOL is used as the programming language.

real: 48 bits. Equivalent to the ALGOL 'real' type.

date: 24 bits. Equivalent to the ALGOL 'integer' type. Assumed to represent a date as a 6 digit integer with the format YYMMDD intended for automatic printing in DUET. However, at present the automatic printing of a date is not yet implemented and DUET will treat a date variable as a word variable.

text: A variable physically quantified in units of 48 bits and intended for storing a text. The length is specified after the type as a decimal point followed by an integer indicating the maximum number of characters in the text excluding the terminating zero character. The type is an ordinary type in DUET, while it is treated as a real array of appropriate length in ALGOL.

recno: 24 bits. Equivalent to the ALGOL 'integer' type. Intended for storing the system generated key (CF record number/BS block position) of a record in a sequential file for a possible later direct access (see section 5.1).

bits: A variable physically quantified in units of 24 bits, primarily intended for anonymous storing of consecutive fields. The length of the variable is specified after the type as a decimal point followed by an integer indicating the number of bytes. The type is an ordinary type in DUET, while it is treated as an integer array of appropriate length in ALGOL.

result: A variable equivalent to the ALGOL 'integer' type. It is intended to communicate the result of some SODA and DUET operations to the program. The kind of result is specified after the type as a decimal point followed by one of the result indicators below.

readterm: After a DUET read operation the value of the terminating character.

readspec: After a DUET read operation the number of the specification that matches the recognized field.

recno: After a SODA NEXT or PUT operation the position of the current record in the file.

soda: The result of any SODA operation.

error: In DUET the identification of an error situation.

Only one variable of each of the above-mentioned result kinds is permitted in an LD description.

The array specification may appear after all types except 'recno' and 'result'. It defines that the variable is an array of the specified type with the number of elements indicated as an integer enclosed in parentheses. An array of type 'text' will appear in ALGOL as a one dimensional real array with a number of elements defined as the product of the real elements needed to represent one text element and the number of text elements.

The optional value spec reference defines by a reference to an entry in the value spectra table (see 3.1.2) the set of relevant values for the variable. Primarily the reference is intended for use in DUET to enable value checking in the read operation.

The optional norm value reference defines by a reference to an entry in the norm value table (see 3.1.3) a standard value for the variable. This value is assigned to the variable by SODA in the CREATE operation if specified so in the declaration of the record set (see 3.2.8). Furthermore the standard values may be assigned to the variable in DUET in connection with a read operation. If the norm value reference is omitted, zero (or an empty text) is regarded as the standard value.

The declaration of a variable as an associated variable by reference to a field is indicated by an equal sign after the colon following the variable name. This must be followed by the name of a record field in the DB description. The variable will then be declared with the type and kind of the referred field. A possible standard value and value spectrum indication for the field will result in corresponding references for the variable to anonymous entries in the norm value and spectra tables. The variable name and the field name may be identical, but it is not required.

When the programming language is ALGOL the SODA LD compiler automatically will generate a textfile containing correct ALGOL declarations equivalent to the variables specified in the variable table. This textfile must then be incorporated in the user program as a part of the ordinary declarations. In DUET, on the other hand, all variables are allocated in a common array and accessed by means of anonymous field addressing. Accordingly the DUET processor, that executes a DUET program, operates directly on the internal representation of the variable table (see ref. 5).

If in a DUET system the surrounding ALGOL program containing the DUET processor need to refer a variable declared in the LD description this is possible without any interference with the variable table. If in this case - i.e. when the programming language is specified as DUET - the user indicates an asterisk (*) after the variable number, the LD compiler will generate an ALGOL declaration of a field variable of the corresponding name and type. The declaration is generated in a textfile to be incorporated in the ALGOL program for the DUET system. The file will furthermore contain a procedure declaration containing for each such indicated variable a statement that assigns the correct address to the field variable. For further explanation of this feature see ref. 5.

Finally a special facility in the LD language concerning operational variable names should be mentioned. In the DUET system the name of a variable is used operationally in two situations:

- in standard error messages from the read operation to indicate which variable was the receiver of a value when the error was recognized.
- in standard layout 'a' in the print operation to produce the name and value of a variable.

When the LD compiler is activated a parameter may specify a non standard language code for the compilation (see 4.2).

The presence of such a code indicates that the user wishes to produce a variable table with a set of alternative variable names to be used by the DUET system in the above mentioned two situations (and only there).

v4	:	stock_on_hand	:	real(2)	w17
(2)		lager_menge			
(3)		varebeholdning			
v6	:		:	date	
v7*	:	order_spec	:	bits.8	
(3)		ordrespecification			

Fig. 3.4 Specification of alternative operational names

The alternative operational names for a given variable which can be selected by the language code are specified in separate lines after the variable declarations as shown in figure 3.4. The appropriate language code is indicated in parentheses followed by the alternative name which should be specified as an identifier.

Alternative names may be specified for all or just for selected variables. Not all relevant language codes with corresponding names need be specified for a given variable. If an alternative name is missing for a selected language code the ordinary variable name will be used.

3.1.1.1 Variable declaration without numbers

The variables can be declared with anonymous variable numbers as shown in figure 3.4a.

```

variables:
  customer_number      word
  *balance             long.2  w6  n2
  vendor_ident        = vendor_name
  soda_ok             result.soda

```

fig. 3.4a: A part of a variable table without variable numbers.

Except the variable number and the colons around the variable name, the syntax is the same as for declaration of numbered variables. But numbered and unnumbered variables can not be declared together in one LD-description.

Variable
reference
by name

All variables can be referred to by their name in the LD description as well as in a DUET program, regardless of how they are declared.

Variable
reference
by number

On the other hand, variable numbers can only be used for variable references, if the numbers are declared explicitly. Indeed, the compiler will generate an internal number for each variable, but this number may change in a recompilation of the LD description.

3.1.2 The value spectra table

As mentioned above a variable may be declared with a reference to an entry in the value spectra table thereby defining the set of relevant values for the variable - the so-called value spectrum. Several variables may refer to the same entry. Entries in the value spectra table may also be referred in the declaration of record sets to express restrictions on the set (see 3.2.2).

The value spectra table consists of a number of lines each one defining a value spectrum. A representative piece of a value spectra table is shown in figure 3.5.

```

value spectra:
w1: r 1 to 15.99
w2: n 5, 3, v7 to -12, 19 to v35, v69
w4: t 5 to 17
w5: r limit1 to limit2, >v33
w6: n -1.000 to 5.999
w7: n .a., .ab., 2000000, .abc., v10
w8: n <0, >3500

```

Fig 3.5 A piece of a value spectra table

The declaration of a variable spectrum consists of an identification followed by a main type and a sequence of single values or intervals.

The identification is formed by a 'w' followed by an integer and a colon.

The main type defines the types of variables which may refer to the entry and indicates how the values of the spectrum should be interpreted or represented according to the following scheme:

- t: The spectrum defines the minimum and maximum number of characters in a text variable. It may only be referred from variables of type 'text'.
- r: The spectrum defines the relevant values of a real variable or field. The constant values of the spectrum are represented as floating point numbers. It may only be referred from variables of type real or from real 'in'-relations of set restrictions (see 3.2.2).
- n: The spectrum defines the relevant values of a variable or field of type word or long with possible implied decimals. It may only be referred from variables or 'in'-relations of these types. All specified elements of the spectrum (see below) should have the same number of decimals, and the variables or fields referring to the spectrum must be declared with this number of decimals also.

The value spectrum itself is a list of elements which are either intervals or single values separated by commas. The list may degrade to a single element. A value used to form the elements may be one of the following kinds:

- a numeric positive or negative constant with possible decimals *)
- a variable number on name referring to a numeric variable
- a so-called short text constant which consists of one to three characters enclosed in decimal points. These characters are packed with their ISO values in 24 bits right justified with possible zeroes to the left.

*) See section 3.1.3 for the limitations of a numerical constant.

3.1.3 The norm value table

A variable may also be declared with a reference to an entry in the norm value table defining a standard value for the variable. Several variables may refer to the same entry. The standard values are used in connection with the SODA CREATE operation (see 5.5) and in DUET in connection with the READ operation.

The norm value table consists of a number of lines each one defining a standard value. A piece of a norm value table is shown in fig. 3.6.

```
norm values:
  n1: n  3.00
  n3: r  v32
  n6: t  'unknown'
  n7: n  customer_number
  n8: r  -1
```

Fig 3.6 A piece of a norm value table

The declaration of a standard value consists of an identification followed by a main type and the standard value. The identification is formed by the letter 'n' followed by an integer and a colon. The main type defines the types of variables which may refer to the entry and how the standard value is represented according to the following scheme.

t: The standard value should be a text constant or text variable, and the variables referring to the entry must also be of the type 'text'.

- r: The standard value should be a decimal number or a variable of type real, and the variables referring to the entry must also be of type real.
- n: The standard value should be a decimal number, a short text constant, or a variable of type word, long or date, and the variables referring to the entry must also be of type word, long or date. A variable referring to an entry of type 'n' must be declared with the same number of decimals as the specified standard value.

The number of digits in a numerical constant (principals and decimals) must not exceed 15, and the greatest possible value is 140 737 488 355 327.

3.2 Declaration of record sets

Access of records by SODA will require that the user declares the necessary record sets in the LD description. The declarations are closely connected to the DB description. It is not possible to declare sets that violate the logical data structure of the database as it is implied in the DB description.

As mentioned in section 2, record sets can be singular or mother subscripted. This must be specified in the head of the declaration and for mother subscripted sets it must be in accordance with the data structures possible from the DB description.

The declarations of record sets are specified after the declarations of variables, value spectra and norm values. This section of the LD description is headed by a single line containing the reserved word 'record sets' followed by a colon.

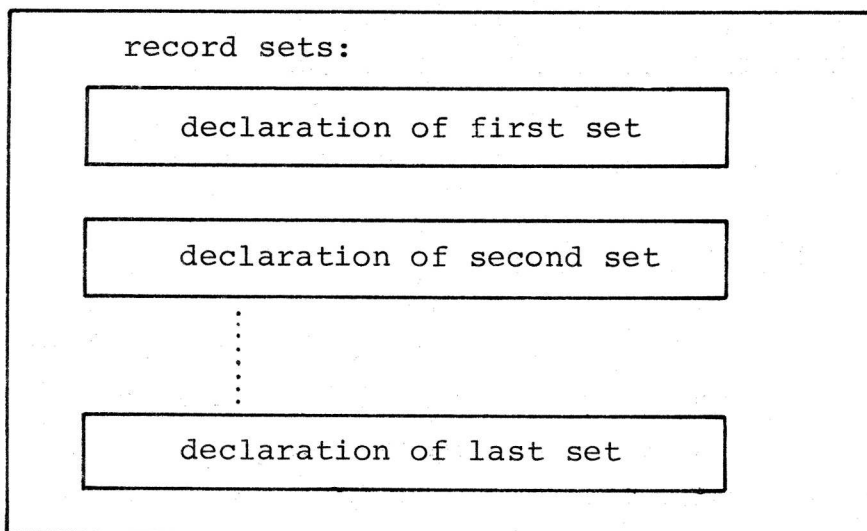


Fig 3.7 General format of the record set section

Each set is declared by a set declaration head followed by a number of specifications, some of which are optional and some only relevant for certain kinds of sets (see below). A 'full' set declaration is shown schematically in fig. 3.8 with the specifications in required order.

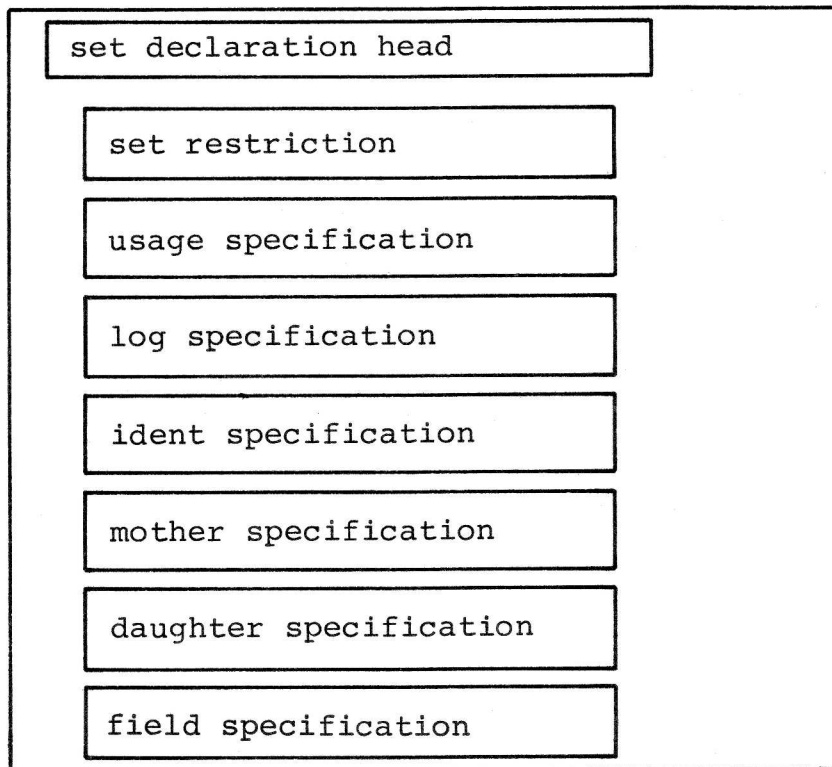


Fig 3.8 General format of a set declaration

3.2.1 The set declaration head

The set declaration is headed by a single line that identifies the set, relates it to a logical file in the DB description and possibly selects a subset of the record types to be the members of the set. Furthermore, in case of mother subscription the line contains a reference to the mother set. Fig 3.9 shows some examples of set declaration heads

```

s2: customers          = customers

s3: sales_items       = products (i17, i18)

s5: spare_parts       = products (items, parts,
                               item_parts)

s6: departements      = depts in org1_file (i58, i59)

s8: special_deliveries (s3) = order_elements (i88)

```

Fig 3.9 Examples of set declaration heads

A set is identified by a set number and an ordinary name. The set number is used in references from other set declarations as for instance the mother set reference in the declaration of s8.

The identification is followed by an equal sign and a reference to a logical file that must be declared in the DB description. If the logical file is declared as a constituent of more than one physical file the reference must be qualified with the name of the physical file as in the declaration of s6.

If all the record types of the logical file are required as members of the set, nothing more need be specified in the head line, as for s2. If, on the other hand, only some of the record types shall belong to the set, this must be indicated by a list of the selected types enclosed in parentheses following the logical file reference as shown for the four other sets. The list may constitute the full set of record types in the logical file but not references to record types declared outside the file.

A record type can be stated as an 'i' followed by the record type number (in s3), or as the record type name (in s5). The record types are separated by a comma and a possible new-line-character.

It is permitted to declare any number of sets referring to the same logical file. Such sets may define any grouping of the record types of the file ranging from a partition, via all combinations of overlapping groups to sets with identical member record types.

If the set is subscribed by mother record this must be indicated in the set declaration head by a reference to the mother set enclosed in parentheses as shown in the declaration of s8 in fig 3.9. The mother set must be declared elsewhere in the LD description but it is insignificant whether it is done before or after the subscribed set.

The mother subscription defines that the subscribed set - the daughter set - is a collection of record sets all referred to by the same set identifier, but each one associated with exactly one record from the mother set. Accordingly, access to one of the subsets of the daughter set can only be obtained by a prior selection of the appropriate mother record. In the application program this is performed by means of the operation NEWSET referring to the daughter set. The current record in the mother set at that moment will then define the selection (see further in 5.7).

For a mother subscribed set the LD compiler will check in the DB description that the records are physically stored in a list file *) (cf ref. 1). Furthermore it will check that the records are physically connected to the records of the mother set. Consistency checks will also be performed according to the information in the associated mother and daughter specifications of the two sets (see 3.2.6-7).

*) The opposite is not the case. It is permitted to declare a set for records in a list file without mother subscription. However only direct access is possible and, in order to utilize this, it is required that the internal record numbers of the records have been stored previously. See 3.2.5 and 5.1.

3.2.2 Set restrictions

Definition of set membership may be further specified for each declared set (mother subscripted or not) by a set restriction. This is a logical expression in variables and constants and values of fields from potential member records of a declared set. The restriction is specified in a separate clause following the set declaration head. If, upon a physical read access of one of the record types defined for the set, the expression becomes true, the record is regarded as a member of the set, otherwise it is not. (see operations GET and NEXT in 5.1-2) Fig. 3.10 shows restrictions for two sets - one defined for all record types of the set and one which is record type dependent.

```
s12: bad_customers    = customers
      for which cu_balance > 10000 and f319 in w7
            or cu_number <= v.min_number

s13: payments (s2)    = transactions (i63, i64, i66, i67, i68)
      for which i63, i66: f119 >= f120 or f130(2) <> 'payment'
            i64: f111 -, in w11
            else : f28=1 and (v12(1)<0.5 or f2 in w8)
```

Fig 3.10 Examples of set restrictions

The logical expression is either a single relation or it is composed of relations separated by 'and' or 'or' operators and parentheses in the usual way. A relation is either a comparison of two values by means of the ordinary relational symbols = <> > < >= <= or it tests for membership or non-membership of a value in a value spectrum defined by a reference to the value spectra table.

The left side operand of a relation is a reference to a field or a variable - either simple or subscripted by a constant. Operands of type bits, aggr, group, recno, mref, or dref are not allowed. The right side operand of an ordinary relation may furthermore be a constant. The two operands should be compatible in type and possible implied number of decimals according to the scheme in fig. 3.11.

right operand → left operand ↘	text var/fld	real var/fld	word.y long.y var/fld	date var/fld	text const	integ. const	decim. const
text var/fld	+	-	-	-	+	-	-
real var/fld	-	+	-	-	-	+	+
word.x } var/fld long.x }	-	-	1)	-	-	+	3)
date var/fld	-	-	-	+	-	2)	-

+ permitted
 - not permitted
 1 permitted if x=y (declared with same number of decimals)
 2 permitted if the constant is < 8388608 (24 bits)
 3 permitted if x>= number of decimals in constant

Fig 3.11 Rules for type correspondance in relations

The comparison of texts will be performed from left to right according to the ISO value of the characters.

Fields and variables in a set restriction can be referred by name or by number. In order to distinguish between a field name and a variable name, the latter must precede by a 'v.' as shown in figure 3.10.

A record type dependent restriction like the one shown in fig 3.10 for s13 contains one or several logical expressions preceded by a record type list or the last one possibly by 'else'. The record type list defines which record types of the set the associated restriction is valid for. The restriction associated with a possible 'else' is then valid for record types of the set not mentioned explicitly. Only record types belonging to the set may appear in the record type lists and no record type may be specified more than once in a set restriction. Furthermore the fields specified in a certain logical expression must be contained in all the record types of the corresponding list. Record types for which no restriction is specified (no 'else' specified) are considered unconditional members of the set.

3.2.3 The usage specification

In a separate clause following immediately after the possible set restriction the user must specify a list of the SODA access operations that in the application program will refer to the current set. The purpose of this statement is primarily to enable the system to check that the various other specifications of the set declaration are consistent with the intended use of the set. Such a check may prevent alarm reactions during the run.

The complete format of the usage specification is shown in fig. 3.12.

```
s2: customers = customers
```

```
usage: next, get, put, create, delete, lookup, newset
```

Fig 3.12. A usage specification

The names of the intended SODA operations must be specified in a single line following the reserved word 'usage' and a colon. The order of the specified operations is insignificant. The specification of LOOKUP and NEWSET is permitted but not required.

3.2.4 The log specification

The fact that SODA hides every physical operation on the database from the user may cause problems in some application systems where it is necessary to track or survey changes in the physical state of the database or the like. Therefore SODA provides a set-specific facility for activating a so-called log procedure when the database is touched. (see 6.3). In a separate clause following the usage specification the user may specify in which situations the log procedure should be activated. Fig. 3.13 shows a representative example of a log specification.

```
s2: customers = customers
    usage: next, get, put, create, delete
    log before: all
    log after : get, next, update, insert, delete
```

Fig 3.13 A log specification

The log specification may consist of one or two lines or may be left out completely. The two lines define the activation of the log procedure just before and just after the file operations involved by the SODA operation, respectively. In both lines either the word 'all' or any combination of the words 'read', 'put', 'newset', 'create', 'get', 'next', 'update', 'insert', or 'delete' may be specified. 'all' means the activation on every operation, 'update' on a 'put' after a read operation and 'insert' on a 'put' after 'create' (see 5.4). Finally the word 'read' means 'get' and 'next', and the word 'put' means 'update' and 'insert'. The other keywords refer directly to the SODA operations. The log specification should be consistent with the usage specification.

3.2.5 The ident specification

The identification of records in SODA operations for direct access and for insertion of such records, is not transmitted by means of explicit parameters. Instead the set declaration must contain a specification of how the necessary values of the ident (or key) fields are derived. This specification is then common for all direct access operations referring to the set, i.e. GET (5.1) and CREATE (5.5). Furthermore it is utilized for positioning in sets where a sequential access should start with a certain record, see NEWSET (5.7).

The ident specification is required for sets if 'get' appears in the usage specification or if 'create' appears and the set is associated with a logical file of type 'CF master' (see ref. 2). It must be stated after a possible log specification as shown in fig 3.8.

Fig 3.14 shows two examples of ident specifications.

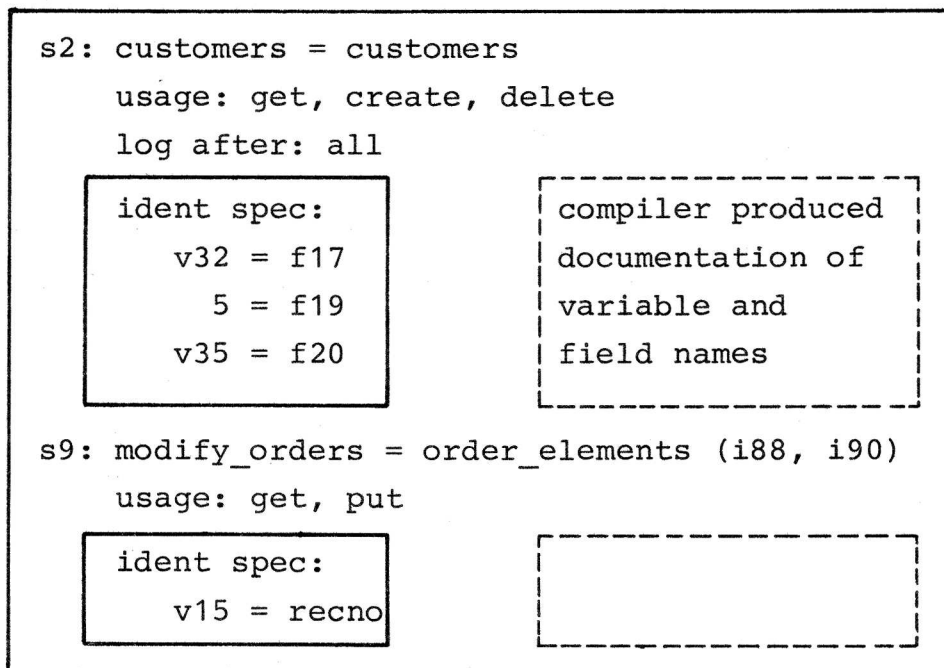


Fig 3.14 Ident specifications in two sets

The ident specification is headed by a line with the words 'ident spec' and a colon. In the normal form (shown in the figure for s2) this is followed by one or more lines each one associating a numeric constant or a SODA variable with an ident field (cf ref. 2). The variable is referred by the variable number or variable name from the variable table and the ident field by the field number or field name from the DB description. The association symbol is an equal sign. When the association lines are read the LD compiler will automatically supply the corresponding variable and field names in the listing as more comprehensible documentation. The normal form is applied in sets associated with master files.

In order to define a proper identification each ident field must appear once and once only in association lines of the ident specification. Moreover for the operands of each association the types must be compatible according to the table in fig 3.15.

left operand	right operand
word var date var long var num constant	byte field word field long field
real var num constant	real field
text var	text field
bits var	aggr field
recno var	'recno'

Fig 3.15 Compatibility rules for operands in ident spec.

In case of 'byte' and 'word' fields the constant or the contents at runtime of the variable at the left side may not exceed the value range for the field type.

An possible implied number of decimals for a variable must match that of the corresponding field.

A field of the type 'aggr' should be associated with a variable of the type 'bits', but if it is declared with a specification of its component fields, a complete set of associations for the components may replace the one for the whole aggregate. Note, however, that only one of the two possibilities may be applied for a given aggregate in a set.

A field array should be associated with a variable declared as array with a number of elements not less than that of the field. The types of the arrays should obey the same rules as those of simple elements.

A special form of the ident specification is applied when a set is used for direct access of records stored in purely sequential files such as a list file (see ref. 1) or a BS file (see ref. 4). In this case the presence of possible ident fields in the records will not influence the access.

Instead an internal key - the record number *) - serve as the identification. This is indicated in the ident specification by a single association line with the word 'recno' instead of a field reference as shown in fig. 3.14 for s9. In this case the variable must be declared with type 'recno' (cf. 3.1.1).

*) see page 2-10

3.2.6 The mother specification

Warning for first time readers: Skip this section until you are familiar with section 2, 3.2.5 and 3.2.8. Even then some cross-reading to the description of CREATE, NEXT and GET (section 5) may be necessary. Anyway be prepared: The whole matter is complicated - and will possible remain so until you really understand all the aspects of the mother/daughter relationship.

One important characteristic of daughter records (see 2.2) is that they have no user-specified ident fields that can serve as an identification suited for direct access. Daughter records are in some sense identified by the mother records they are connected to. In the LD description this is reflected in the mother specification. It defines a communication of values of m-ref fields from SODA to the application program and/or vice versa. In this way the mother specification can be regarded as a special kind of a field specification or ident specification respectively, depending on whether it is used when a daughter record is read from the database or created as a new one.

When a daughter record is read the values of its ordinary fields are made available for the program in variables according to the variable/field associations in the field specification (see 3.2.8). However, if one or more of the values of an m-ref field are needed the problem arises that the DB description does not provide local names for the possible sub-fields of an m-ref field - those which are direct pictures of the ident fields in the corresponding mother record. (see fig 3.16).

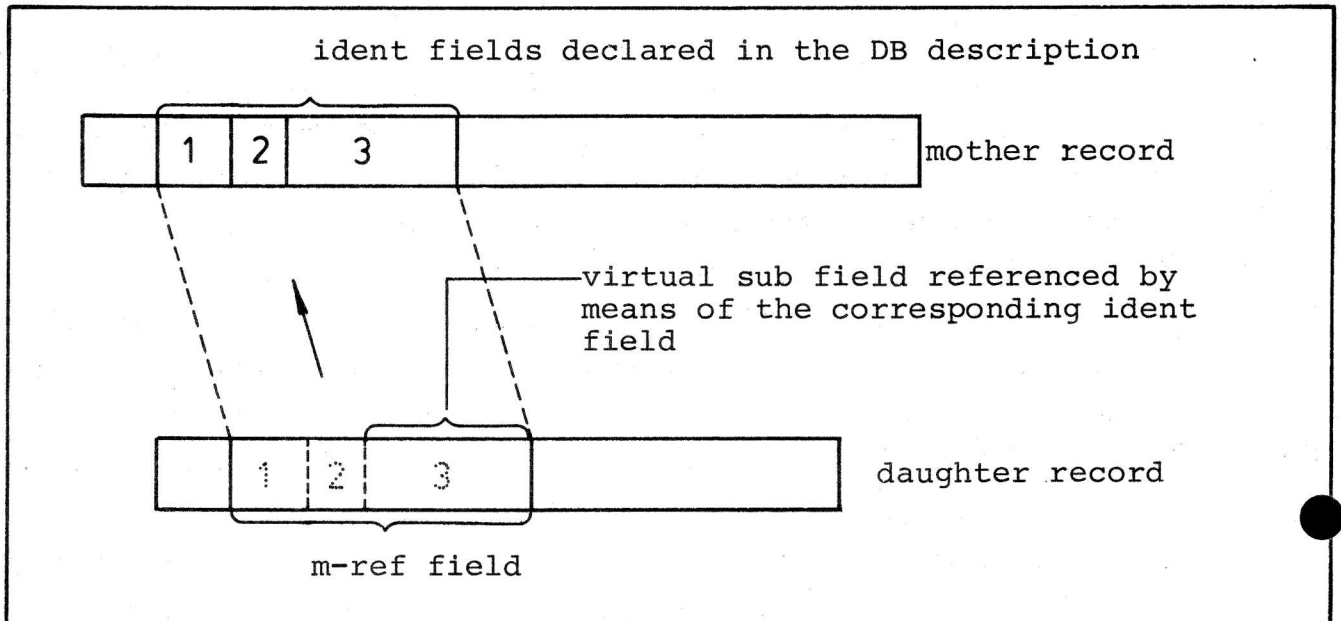


Fig 3.16 An m-ref field mapping the set of ident fields in the mother record.

For this reason it is necessary to use references to the ident fields in the mother record when the (virtual) subfields of an m-ref field are associated with variables. The associations are then grouped so that the appropriate m-ref fields appear as a common qualifier for all fields in the associations of the group. (cf. fig 3.18).

When a daughter record is created by SODA it can only be done in a mother subscripted set. If now the daughter record (in the DB description) is declared to be connected by one link only then its insertion in the database is well defined as soon as it by CREATE is established as the current record of the daughter set. At the following PUT operation it will be inserted in the database linked to the subscripting mother record. However, if it is declared as connected by more than one link to other records the implicit reference to the subscripting record of the mother set alone will not do it.

The CREATE/PUT operation will need the identification of the secondary mother records too. This is derived from information in the mother specification which for this purpose must associate a variable (or a constant) with each virtual sub field of the ('non subscripting') m-ref fields. (see fig 3.17).

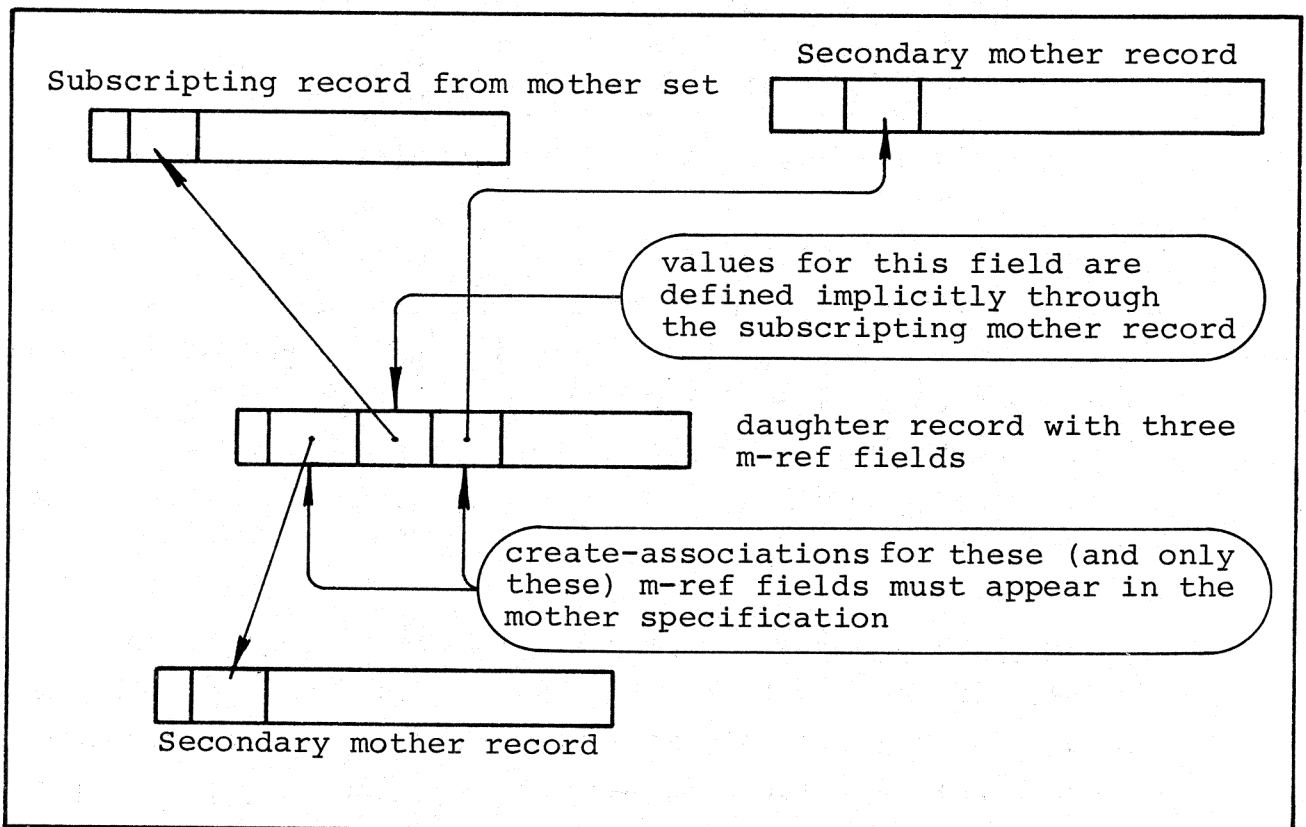


Fig. 3.17 Derivation of values for m-ref fields when a daughter record is created in a mother subscripted set.

The mother specification must appear after a possible ident specification as shown in fig 3.8. It is composed of one or more entries each one headed by a single line with the word 'mspec' followed by a colon and the field number of an m-ref field. For mother subscripted sets an entry for the m-ref field corresponding to the subscribing record is not allowed. Fig 3.18 shows the structure of a mother specification.

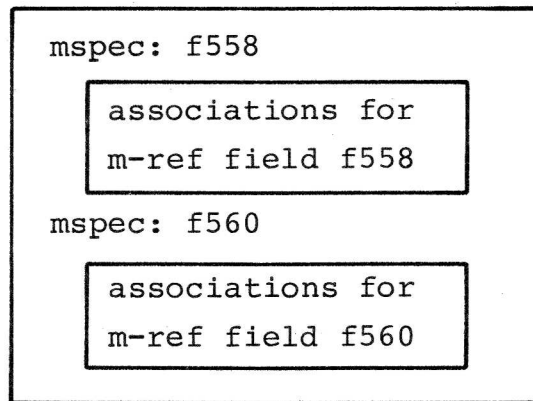


Fig 3.18 General format of mother specification.

The association lines of each mspec entry is composed of a variable reference (or a numeric constant), an association symbol, and a reference to an ident field in the mother record corresponding to the m-ref field. The association symbol defines in which direction the values are transferred and thereby the SODA operation by which the transfer is performed (see fig 3.19).

The association symbol < declares that the value of the specified sub-field in the m-ref field should be moved to the specified variable (no constant allowed) whenever a read operation on the set results in the establishment of a current record for the set.

The association symbol -> declares that the value of the specified variable or constant should be used in CREATE. Here it serves as (a part of) the identification of the mother record to which the daughter record should be linked when the following PUT is activated. At this time the value will be transferred also to the appropriate subfield of the m-ref field in the daughter record.

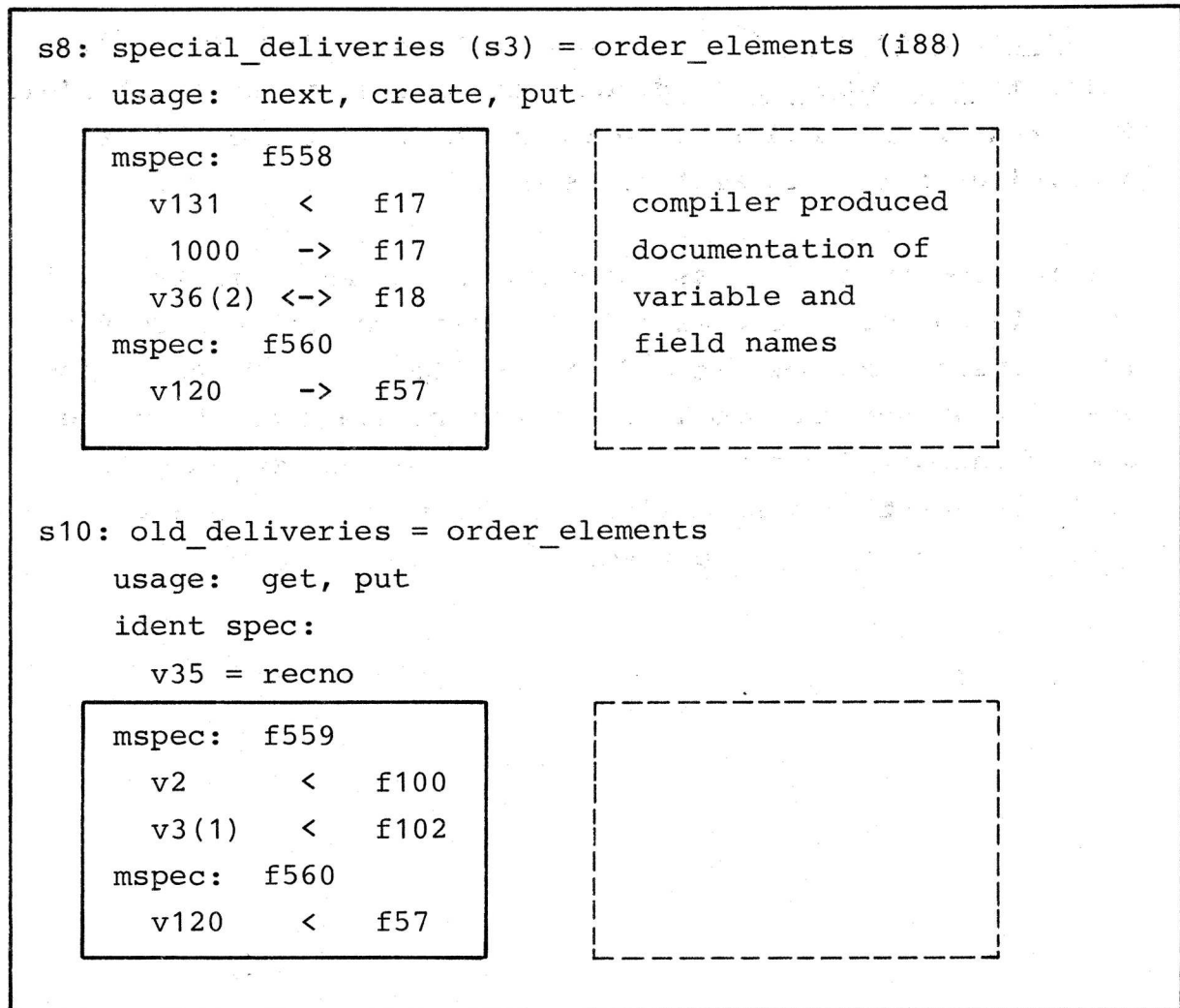


Fig 3.19. Two examples of mother specifications

The association symbol <-> combines the function of < and -> and defines a transfer between the two operands both on a read operation and on a CREATE/PUT operation. A constant is not allowed in connection with this symbol.

The variable reference is a variable number possibly followed by a constant subscript if the variable is declared as an array. The LD compiler will supply the name of the variable and of the field as an automatically generated comment in the same line of the listing.

The field reference is a field number referring to an ident field in the mother record, or more precisely: an ident field declared in the logical file to which the m-ref is linked according to its declaration (see ref. 2).

The two operands of an association should match in type and possible number of decimals in the same way as defined for ident specifications (see 3.25 and fig 3.15). Moreover the association symbols should be in accordance with the usage specification. The symbols < and <-> are allowed only if 'get' or 'next' are specified and the symbols -> and <-> only if 'create' is specified.

3.2.7 The daughter specification

As mentioned in section 2.3 access of record fields representing attribute properties is accomplished via variables of compatible types. The variables are linked to the fields in a set of variable/fields associations in the field specification (see 3.2.8). For the fields representing relational properties of the 1:n-type - the so-called d-ref fields^{x)} - such associations have no meaning. From the user's point of view their 'value' should rather be regarded as the whole collection of records that are linked to the one containing the relational field.

Therefore, it is more sensible to associate such a field with a record set, and since it expresses a mother/daughter relation it should be a mother subscripted set. An association of this kind then declares that when the mother record is at hand the user wish to access the value of the relational field - that is, the daughter records - just like the ordinary associations indicate access to values of attribute fields. The difference is that values of attribute fields become available in variables while the records must be accessed one at a time after an introductory activation of NEWSSET.

The associations appear in the daughter specification as simple lines following a common head line consisting of the word 'dspec' followed by a colon. The daughter specification should be stated before a possible field specification. Each association is composed of a reference to a mother subscripted set declared elsewhere in the LD description, the association

x) In the Database 80 language (see ref. 2) d-ref fields are declared as type 'list'

symbol <, and a reference to a field declared in the DB description as a d-ref field in at least one of the record types of the set. Fig. 3.20 shows a daughter specification.

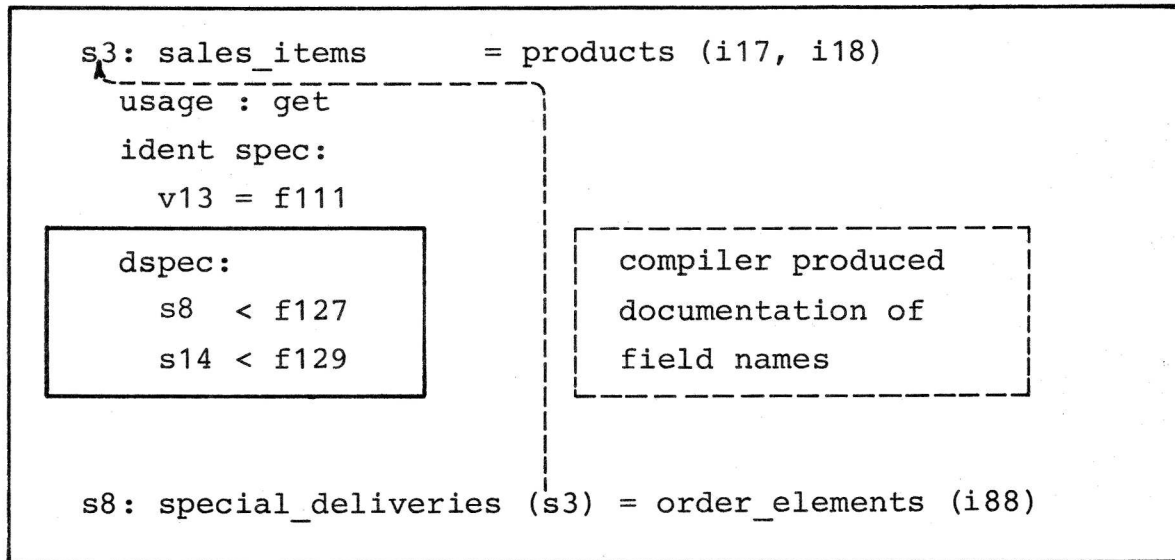


Fig 3.20 A daughter specification illustrating the relation between mother and daughter sets

The daughter specification is used for another purpose too, which is related to the deletion of mother records. SODA requires that when a record is deleted, that according to the DB description could be a mother record, it should not have any daughter records linked to it. (If there are any, they have to be deleted previously). This is checked by SODA and an alarm reaction will occur if the mother record actually is linked to any daughter records. On the other hand, SODA enables the user to delete freely any such potential mother record that satisfies the above-mentioned condition, also when none of its potential daughter record types are declared as members of sets and whereby no operations can be performed on them in the program.

In order to remind the user of these conditions, SODA requires that all d-ref fields belonging to the member records should be referenced once in the daughter specification for a set where 'delete' is indicated in the usage specification. In such a situation it is not necessary to associate the d-ref fields with mother subscripted sets, but each field reference should occupy its own line in the daughter specification. Fig. 3.21 shows an example of such a daughter specification.

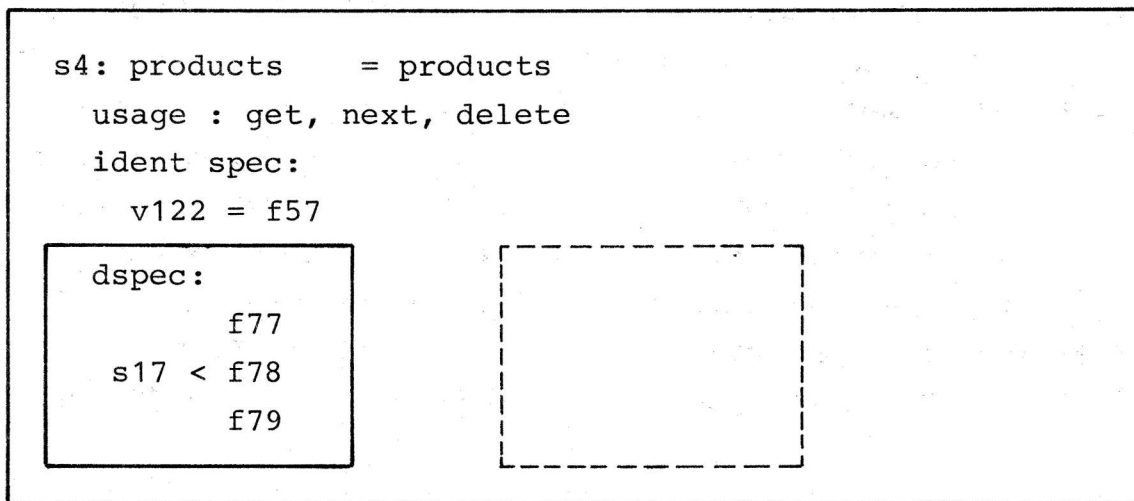


fig. 3.21 A daughter specification with all d-ref fields indicated, some without association to a daughter set.

3.2.8 The field specification

In order to access the ordinary record fields - i.e. fields representing attribute properties - the user must specify a list of field/variable-associations organized in the last kind of entries in the set declaration - the field specification. The field specification should appear after possible ident, mother and daughter specifications in the set.

The associations specify the relevant fields of the record in the set and how the application program wishes to access them. By means of the association operator it is indicated whether it is just the value after a read operation, or whether the program wishes to insert its own value in a field before the record is returned to the database, or possibly both. It can also be indicated whether the insertion of a new field value should be performed on all put operations, or exclusively on those following a read operation or those following a CREATE operation.

The associations should be grouped so that each group is headed by a separate line. It contains the reserved word 'fieldspec' followed by an integer and a colon. If the integer is greater than zero it serves as an identifier of the group and must be unique inside the set. Each group may contain up to 63 individual associations *).

*) The purpose of this grouping is primarily to enable the introduction of a copy operator to be applied in the LD description when the same set of associations are relevant in more than one set. The copy operator will then refer to the group by its number.

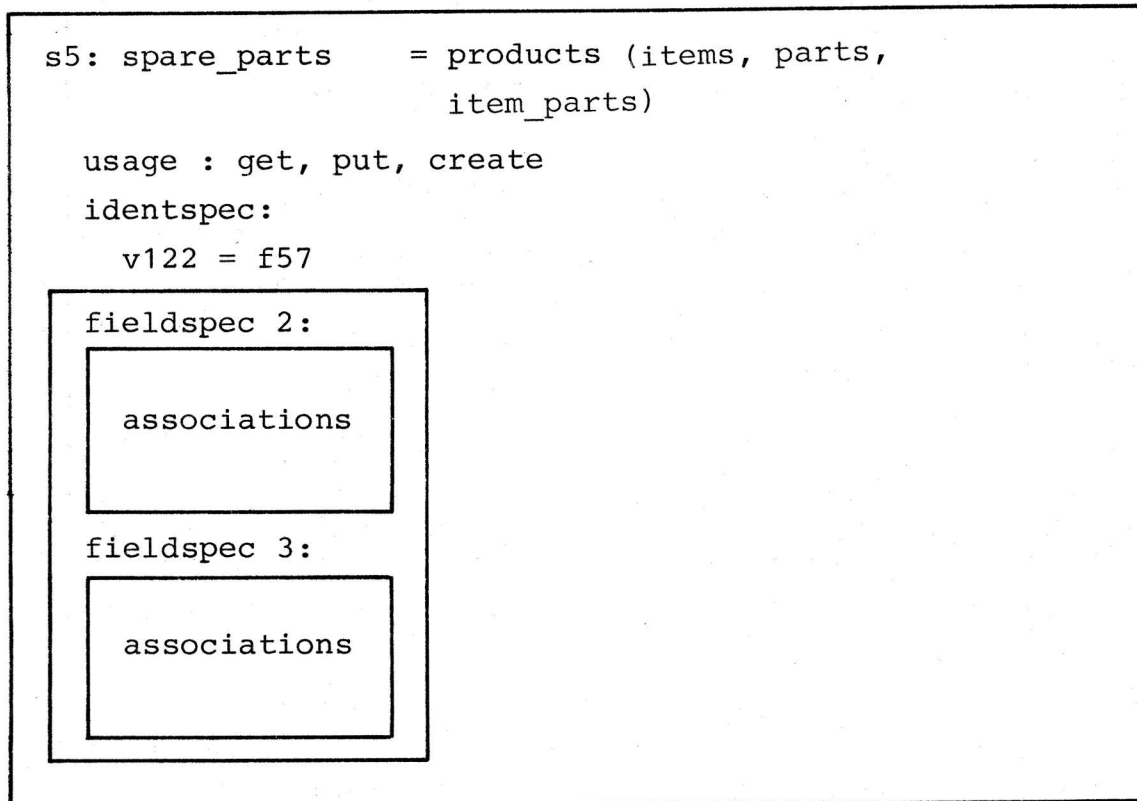


Fig. 3.22 Two groups of associations forming a field specification

An association is composed of:

- a left part containing either a numeric constant or a variable reference
- an association operator
- a right part containing a field reference or the reserved word 'recno'.

Not all combinations of these elements are meaningful. Fig. 3.23 shows a representative set of valid associations.

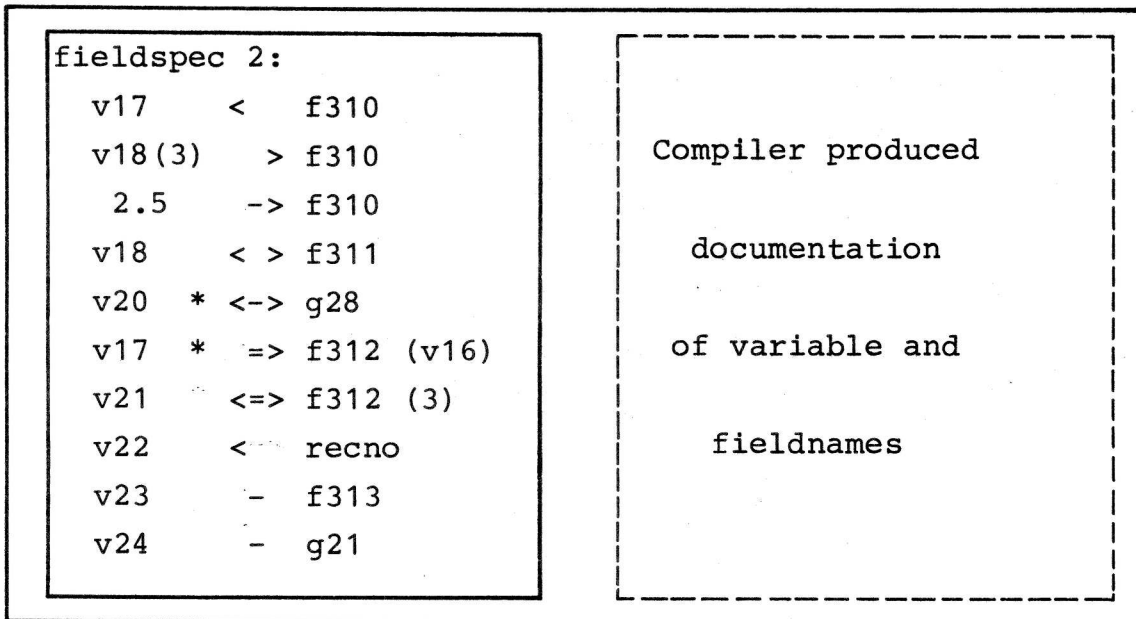


Fig. 3.23 A representative set of associations in a field specification

The various possibilities and the rules to apply for composing associations concern the association operator and the type of the two operands. The description below follows this scheme, but the great number of possible combinations resulting from the intended flexibility may cause the description to appear rather instructed.

The association operator defines both a direction of transfer and when the transfers are executed. Each association is checked against the usage specification so that it is impossible, for example, to specify a transfer from variable to a field without a usage PUT.

The operator < (to be interpreted as an arrow) defines that upon a GET or NEXT operation the value of the specified field in the current record of the set is moved to the variable specified in the left part. If the field is not present in the current record, no transfer will take place. The same field may be moved to more than one variable, but a given variable may not be specified in a set to receive values from more than one field, unless the fields are exclusive with respect to record type^{x)}.

The operator > defines that before a current record obtained by GET or NEXT (but not by CREATE) is transferred back to the database by PUT, the value of the left part variable or constant is moved to the location of the specified field. If the field is not contained in the current record no transfer will take place. The same variable or constant may be moved to more than one field, but a given field may not be specified to receive values from more than one source^{x)}.

The operator -> works like the operator > except that the transfer is performed only when a current record obtained by CREATE is inserted in the database and PUT.

The operator => combines the function of the operators > and -> so that the transfer to the field is performed on all PUT operations.

It is possible in the same association to combine the operator < with any of the operators >, ->, or => resulting in transfers in both directions. Thus, in the example in fig. 3.23 the value of f311 will be moved to v18 upon a GET while the value of v18 will be moved to f311 on a following PUT.

x) At present and until further notice this is not checked by the compiler.

The operator * may be specified in front of association operators containing either -> or => if the left part is a variable. The presence of an * will not influence the transfers defined by the association operator. It defines that on a CREATE operation the standard value (from the norm value table) will be assigned to the variable. If no such value is specified, zero or an empty text is assumed as the standard value.

The operator - requires that the left part is a variable. It defines a transfer to the variable upon a GET or NEXT operation. If the specified field is present in the current record - formally spoken : declared as a member of the record type - then the record type value is assigned to the variable. If the field is not present zero will be assigned.

The left part of the association is normally a variable reference, but it may be a numeric constant if the operator defines a transfer to a field only and not the opposite way. A variable is referred by means of the variable number⁺). In the listing the LD compiler will supply the variable name. The reference may specify:

- a simple variable
- a whole array
- an array element (constant index only)

The variables must be declared in the variable table and their types should be compatible with the types of the corresponding fields. A survey of the rules for the type match is given in Fig. 3.24-25.

The right part of the association is normally a field reference, but it may be the reserved word 'recno' if the association operator is <, indicating a transfer of a system generated key value. A field is referred to by means of the field number (or group number) from the DB description⁺). In the listing the LD compiler will supply the field name together with the name of the variable in the association.

⁺) It is possible to specify the names of the variable and/or the field of an association in stead of the numbers, in which case the numbers are supplied as compiler generated documentation. In order to obtain a nice listing, however, one should only use one reference type inside a set declaration.

The field reference may specify:

- a simple field
 - a field array
 - an element of a field array (constant or variable index)
 - a non repeating group (group or aggregate)
 - a repeating group
 - a repeating group vector (see next page)
 - an element of a repeating group vector (variable index only)
- } referred to by
group-number

Each referred field must be declared in the DB description for at least one of the record types of the set. The type of the field should be compatible with the type of the corresponding left part of the association according to the survey in fig. 3.24 and 3.25.

The values to be transferred depend on the type of the two operands. The following description covers the various kinds of transfers. (cf. fig. 3.24-25).

Simple values: This case refer to associations where the left part is a simple variable, an array element, a numeric constant, or a short text (see 3.1.2), and the right part is a simple field or an element of a field array or repeating group vector. The rules depend on the type of the value:

Numeric: A real value can be moved to a real location and nowhere else. A non real value can be moved to a non real location. A decimal constant must match a real field and an integer constant or a short text a non real field. The values are moved irrespectively of any implied decimals, but the system secures that a value will not exceed the range of a receiving field.

Date: A date value can be moved to a date location and nowhere else.

Text: The length of the text must not exceed the length of the receiving location

Group/aggr/bits: The length of the value must not exceed the length of the receiving location. The bits are moved irrespectively of a possible internal type specific structure

Arrays: If both the left and right part are arrays (not elements of arrays) the contents of the whole array is transferred. The receiving array should have at least the same number of elements as the sending array. The types should match as for simple values.

If the left part is a constant its value is transferred to all elements of a numeric field array. Again, here, decimal constants to real fields and integer constants or short texts to non real fields.

Repeating group vectors: A repeating group (rpg) vector is an element of a repeating group. It can be regarded as an array with a variable number of elements specified by a value associated with the rpg itself (and stored in a separate location in the group). The number of repetitions is defined and fixed when the record is created. Accordingly, rpg vectors are treated in most respects as arrays with rules as mentioned above. The only exceptions are that the number of elements transferred is decided dynamically and that a variable array in the left part should have at least the same number of elements as the maximum number of repetitions for the rpg.

Repeating group number of repetitions: The right part should be a group number referring to a repeating group. An association of this kind is relevant in the following two situations:

Creation of a record containing the rpg: The number of repetitions is defined by the left part of an association with the operator ->. After the creation the number of repetitions cannot be changed.

Reading of a record containing the rpg: The actual number of repetitions of the rpg in the current record is transferred to the variable in the left part of the association.

System generated record keys (record numbers): This case refers to associations with < as the operator and 'recno' as the right part. Such associations are only legal for sets associated with BS files or CF list files where direct access by means of values of user specified ident fields cannot be accomplished. When records from such files are read by SODA, an association as mentioned will supply a system generated key value in the left part variable (of type 'recno'). This value can be used for direct access of the record at another time (see 3.2.5).

NOTE: The associations belonging to field specifications for a given set will be 'executed' in lexicographical order. In this way it is possible to get a value transferred from a record field - a value which may be used to select an element of an array or repeating group vector present in the same record.

	right part		simple or array element												
	byte	word	long	date	real	text	group	num. array	text array	num rpg. vector	text rpg. vector	rep. group	'recno'		
word	1	1	2	0	0	0	0	0	0	0	0	11	0		
long	1	1	1	0	0	0	0	0	0	0	0	0	0		
date	0	0	0	1	0	0	0	0	0	0	0	0	0		
real	0	0	0	0	1	0	0	0	0	0	0	0	0		
text	0	0	0	0	0	1,4	0	0	0	0	0	0	0		
bits	0	0	0	0	0	0	1,4	0	0	0	0	0	0		
recno	0	1	0	0	0	0	0	0	0	0	0	0	3		
num array	0	0	0	0	0	0	0	5,6,8	0	8,9 10	0	0	0		
text array	0	0	0	0	0	0	0	0	5,6, 7	0	7,9 10	0	0		

- 0: Illegal operand combination
 1: Transfer of whole value
 2: Transfer of rightmost 24 bits of value
 3: Transfer of system generated key
 4: Length of field may not exceed length of variable
 5: Transfer of whole array
 6: Number of elements in variable may not be less than number of elements in field
 7: Length of element in field and variable must be equal
 8: Types of operands must match as for simple values
 9: Transfer of the actual number of elements in the rpg. vector
 10: Number of elements in variable may not be less than the declared maximum number of repetitions in the rpg.
 11: Transfer of the actual number of repetitions (= number of elements in the vectors)

fig 3.24

left part		right part											
		byte	word	long	date	real	text	group	num. array	text array	num rpg. vector	text rpg. vector	repeting group
simple or array element	word	2	1	1	0	0	0	0	0	0	0	0	3
	long	2	2	1	0	0	0	0	0	0	0	0	0
	date	0	0	0	1	0	0	0	0	0	0	0	0
	real	0	0	0	0	1	0	0	0	0	0	0	0
	text	0	0	0	0	0	1,4	0	0	0	0	0	0
	bits	0	0	0	0	0	0	1,4	0	0	0	0	0
	recno	0	1	0	0	0	0	0	0	0	0	0	0
integer constant/ short text		2	2	2	2	0	0	0	8,11	0	8,11	0	3
real constant		0	0	0	0	1	0	0	8,11	0	8,11	0	0
num array		0	0	0	0	0	0	0	5,6, 8	0	8,9, 10	0	0
text array		0	0	0	0	0	0	0	0	5,6, 7	0	7,9, 10	0

Fig 3.25

- 0: Illegal operand combination
- 1: Transfer of the whole value
- 2: Transfer of value if it does not exceed the range of the receiving field
- 3: Only permitted in connection with the association -> Defines the actual number of repetitions of the group to be reserved in the created record
- 4: Length of variable may not exceed length of field
- 5: Transfer of the whole array
- 6: Number of elements in field may not be less than number of elements in variable
- 7: Length of element in field and variable must be equal

Fig. 3.25 continued on next page

- 8: Types of operands must match as for simple values
- 9: Transfer of the actual number of elements in the rpg. vector
- 10: Number of elements in the variable may not be less than the declared maximum number of repetitions in the rpg.
- 11: Transfer of the value to all elements of the array or to the actual number of elements in the rpg. vector.

3.3 Automatical declaration of record sets

By means of the reserved words 'record output' and 'record input' you can declare in the LD description two record sets to be used for creation of, and for sequential reading from a bs-file, respectively. The declaration of these sets will automatically generate variable declarations and field-variable-associations. These sets must be declared before the ordinary record sets, and they are only allowed, if variables are declared without numbers.

3.3.1 Record output

Figure 3.26 shows the declaration of a set for creating a bs-file.

```
record output: transfile
               transactions (i17, i18)
```

fig. 3.26: Example of a record output set.

The compiler will on this declaration simulate a set declaration like the one shown on figure 3.27.

```
s1: trans = transactions in transfile (i17,i18)
    usage: create put
```

fig. 3.27: How a record output set is interpreted.

Sequential
file

Here 'transactions' is the name of the logical file, whereas 'transfile' is the name of the physical file, which is required to be a sequential file (file type = outvar).

Record types

The record types can be omitted, if all record types in the logical file are required as members of the set. If any record types are indicated, a record type can be specified as an 'i' followed by the record type number, or as the record type name. The record types are separated by a comma and a possible newline_ character.

Set number
s1

The set number for a record output set is always s1. This indicates that only one record output set can be declared. On the other hand, this set can, contrary to a normal set declaration, contain more than one logical file as shown in the example in figure 3.28.

```
record output: transfile
  db_copy (items, customers, orders)
  genius_trans (i1, i2, i5, i27, i28,
                i33, i37, i45)
  transactions
```

fig. 3.28: A record output set comprising several logical files.

Implicit
variables

The compiler produces implicit variable declarations corresponding to the fields in all record types belonging to the set. The principles of this variable generation are described in section 3.3.2.

Field association
symbol ->

Furthermore, field associations of the type -> between the variables and the corresponding fields are produced, so that every field will be assigned when a record is inserted into the file.

3.3.2 Implicit variable declarations

Variable name	<p>The variables for a record output set are declared with the same name as the corresponding field, except the prefix *) of the field name, which is removed. So, every field in the DB description used for record output must have a prefix and the first character after the prefix must be a letter in order to obtain a legal variable name.</p>
Type	<p>The variable is declared with the same type, dimension, value spectrum, and norm value as the field.</p> <p>Many fields can be attached to the same variable, as the prefix is removed from the field name. The compiler checks that all these fields have the same type and possible array dimension. The norm value of the fields, must be the same, too, whereas the value spectra of the fields may be different. In this case the value spectrum of the variable is defined as the union of the different field value spectra.</p>

*) The prefix is all characters up to (and including) the first underlined space.

- Explicit and implicit variables
- A variable to be used in the record output set may be declared explicitly in the variable part of the LD description; this is in order to define the value spectrum and the norm value independently of the value references of the fields. These are in this case ignored.
- Associated and implicit variables
- A variable declared as an associated variable in the variable part, however, is completely juxtaposed to the implicit variables concerning the value references.
- Administrative status
- Most of the record fields having an administrative status do not cause a declaration of variables and field associations, as these fields are assigned by the PUT procedure. This holds for the fields with administrative status
- reclength,
checksum,
rectype,
- whereas fields with adm.status
- adate,
ident
- are treated as ordinary fields.
(The other possible administrative fields are of no interest in a sequential file and will be skipped too).
- Copy record
- NB ! If a record type contains a 'copyrecord' all fields from the copied record have no administrative status, i.e. they are recognized as normal fields, even if they were administrative fields in the original record.

Aggregate,
group

A field of the type 'aggr', 'copyaggr', or 'group' (but not a repeating group) is skipped, because only the detailed fields are of interest. This implies that a copy aggregate should not be declared in the DB description without naming of the detailed fields *).

Repeating
group

A field of type 'rpg' will declare a word variable, which shall define the number of repetitions in the actual record. For each element of the repeating group an array variable is declared with a number of elements corresponding to the maximum number of repetitions.

Mref, dref

A field of the type 'mref' or 'dref' can occur only in a copy record. Both are skipped, but following an mref field, the ident fields of the mother record are placed. These fields will be associated with variables declared with special names constructed as

mref	listno	_	mother field name
------	--------	---	-------------------

'listno' is the list number defined by the list name in the mref field. 'mother field name' is the field name of the mother field, the prefix included.

*) At present this is not checked by the compiler.

Equivalence
field

If an equivalence field is used for naming an array element, the SODALD compiler generates variable declarations and field associations for the array field as well as for the equivalence field. The equivalence variable is declared as a simple variable, but with the same type and value reference as the array.

The array variable should never be used by your program, as the transfer from the equivalence variable will be executed after the transfer from the array variable.

Listing of
record type

The LD compiler produces to each record type a listing of all fields, their types and value references, and - if a variable is attached to a field - the variable name. Figure 3.29 shows an example of this listing, based upon the example 11 in DATABASE80 (ref.2).

EAH	17.03.1978 - 17.05	1 / MANUAL	SODA - LD
		RECORD OUTPUT	SYSDOK: MANUALEKS 3.
			VERSION: 7
I1402 TRAVEL←COPY:			
F1 U20←RECLENGTH	WORD		
F2 U20←CHECKSUM	WORD		
F3 U20←RECTYPE	WORD		
F4 U20←ADATE	WORD		* ADATE
F5 U20←GENIUSER	WORD		* GENIUSER
F6 U20←SA←KEY	AGGR.4		
F7 U20←CASENO	WORD	VREF 1	* CASENO
F8 U20←CASETYPE	WORD	VREF 2	* CASETYPE
F9 U20←TYPE	WORD		* TYPE
F1 FB←RECLENGTH	WORD		* RECLENGTH
F2 FB←REC←NO←CF	WORD		* REC←NO←CF
F3 FB←RECTYPE	WORD		* RECTYPE
F4 FB←ADATA	WORD		* ADATA
F5 FB←RECSTATE	WORD		* RECSTATE
F6 FB←FSTATE	WORD		* FSTATE
F90 SA←CONSUMPT	BYTE(4)		
F10 SA←KEY	MREF(1)		
F11 SA←CASENO	AGGR.4		
F12 SA←TYPE	WORD	VREF 1	* MREF1←SA←CASENO
F91 FB←IDENT	WORD	VREF 2	* MREF1←SA←TYPE
F2001 FB←KM←AMT	WORD		* IDENT
F2002 FB←SUBST←AMT	LONG.2	VREF 8	* KM←AMT
F2003 FB←OTHER←AMT	LONG.2	VREF 8	* SUBST←AMT
G402 VOUCHERINF	LONG.2	VREF 8	* OTHER←AMT
F201 FB←VOUCHERNO	FIELDGR.4		
F202 FB←VOUCHERDATE	WORD	VREF 5	* VOUCHERNO
G401 EMPLOYEEREF	DATE		* VOUCHERDATE
F101 MA←CONSUMPT	FIELDGR		
F10 NA←KEY	MREF(2)		
F11 NA←EMPLOYEEENO	AGGR.4		
	LONG	VREF 3	* MREF2←MA←EMPLOYEEENO

fig. 3.29: Example of the listing of a record type in a record output set.

3.3.3 Record Input

A set intended for sequential reading from a bs-file can be declared as a record input set as shown in figure 3.30.

```
record input:  inputfile
               db_copy (items, customers, orders)
```

fig. 3.30: Example of a record input set.

The compiler will on this declaration simulate a set declaration like the one shown in figure 3.31.

```
s2: intrans = db_copy in inputfile (items,
                                   customers, orders)
usage: next
```

fig. 3.31: How a record input set is interpreted.

'db_copy' is the name of the logical file, and 'inputfile' the name of the physical file, which must be a sequential file.

The compiler generates implicit variable declarations and field associations in the same way as described for record output sets (section 3.3.1 and 3.3.2).

There are a few differences in the generation of variables and field associations, as listed below:

- | | |
|------------------------------------|--|
| Association
symbol < | - The field association symbol is <, i.e. every variable connected to the record will be assigned when a record is read from the file by NEXT. |
| No value spectra
and norm value | - Implicit variables are declared without any value spectra and norm values. (But if the variable already exists with a value spectrum or a norm value, these will of course remain). |
| Administrative
field status | - The fields with administrative status
reclength,
rectype
are treated as ordinary fields, i.e. variable declarations and field associations are generated. So, if a record contains a 'copyrecord', you must provide for different names - after the prefix - to these fields in the new record and the copied record. |

3.4 Connection to the Data Base Description

Normally, the LD description is referring to only one DB description. A record output set or a record input set may, however, refer to another DB description but the ordinary record sets.

Change of
DB descr.

A change of DB description file is specified by the line

```
descripfile = db_file_index
```

where db_file_index must be a number 1, 2, or 3, referring to the descripfile names specified in the activation parameters (cf. section 4.1.1).

The descripfile line can be specified immediately before each of the following lines:

```
variables:  
record output:  
record input:  
record sets:
```

When no descripfile has been specified, 'descripfile = 1' is implied. After a specification of a descripfile, this DB file is used until a new descripfile is stated.

If more than one DB file is used by the LD description, you must specify in the activation parameters, which one is going to contain the compiled LD file.

4. The SODA LD compiler

This section describes the compiler for the SODA LD language, its main function, how it is activated, and the different results it may produce.

Readers of this section are assumed to be familiar with section 2 and 3 and to have a common knowledge of how programs are activated on RC4000/8000 under BOSS and the file processor (ref. 9).

Main function

The purpose of the SODA LD compiler is to read a SODA LD description, check it for correctness and produce a file with the information necessary for the DBMS to operate on the DB.

Input:

The compiler requires two input files - the LD description and the DB description file.

LD text

The LD description may either be an ordinary text file or it may be represented in a SYSDOK file.

DB description file

The DB description file contains the information from a correct DB description as produced by the Database80 compiler.

Output:

The compiler may produce up to four output files three of which are optional or depend on the result of the compilation. The output files are:

- Standard output with log and possible error list

- Listing of the LD description (optional)
- Text file with generated declarations
(optional/result dep.)
- LD file with information to the DBMS
(optional/result dep.)

The log

The log contains a documentation of the compilation with identification and description of the files involved and produced. Details are explained in 4.2.

The listing

The listing, which is produced on demand only, contains a complete reproduction of the LD text as read from the input, but edited in format to improve the readability and supplemented with compiler generated comments. Details are explained in 4.2.

The declaration
file

This text file will be produced if the programming language for the application program is ALGOL, or, in case of DUET, if so specified in the variable declarations of the LD description (see 3.1.1). The text must be incorporated in the ALGOL program before it is compiled as explained in 6.1 and 6.2.

The LD file

This file will be produced on demand if the LD description is formally correct. It contains an internal representation of the information from the LD description and the referred parts of the DB description which are necessary for the DBMS. Furthermore the file is used by the DUET compiler, primarily because it contains information about all the declared variables to be used in the DUET program.

The file is identified with information both from the LD text and from the activation parameters for the LD compiler as documented in the log. The LD file is stored in the same physical file as the DB description file.

Error messages

If the compiler recognizes formal errors each one will be communicated to the user as an error message. Errors concerning the activation of the compiler will appear on standard output as described in 4.2 while errors concerning the LD description will appear both on standard output and on a possible listing (see 4.3 and appendix B).

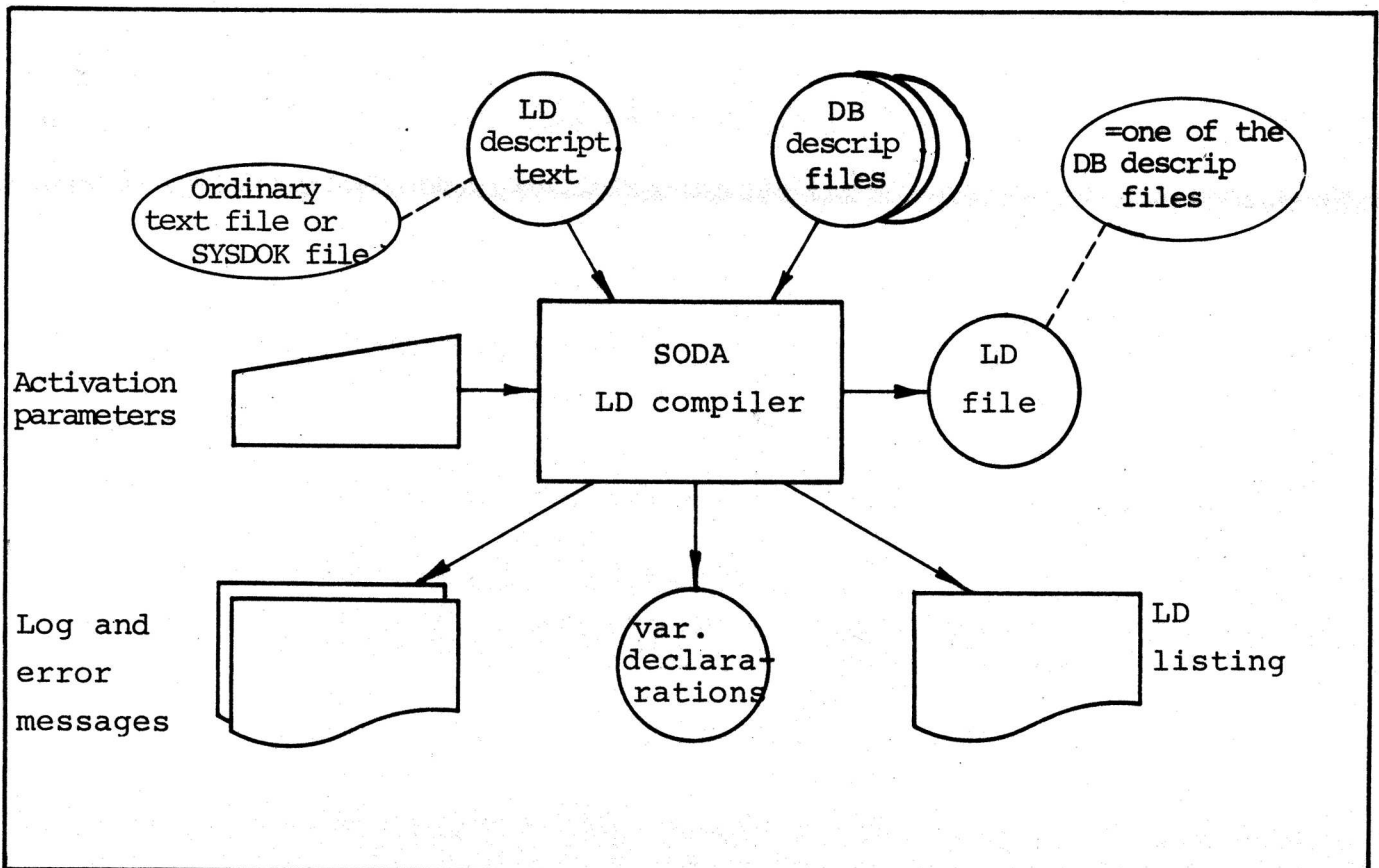


Fig. 4.1 Input to and output from the SODA LD compiler

The LD compiler is normally activated as any other program on RC4000/8000, that is, as a part of the BOSS job with the necessary information to control its function represented as FP parameters. In order to obtain a reasonable compile time for realistic LD descriptions the size of the process should be at least 150000 bytes. *)

The data specifying an activation must obey the general FP syntax rules (see ref. 9), that is, it is the name of the compiler 'sodald' followed by a list of parameter groups which together may occupy one or more lines. Each parameter group consists of a keyword possibly followed by a number of specifying parameters separated by points.

Figure 4.2 contains a survey of all parameter key-words.

Most of the parameter groups can be left out, in which case standard values are used, as described for each parameter group. If a parameter group is mentioned more than once, the last will be valid.

Figure 4.3 at the end of this section shows two examples with explanation of activation commands.

Below is a description of each parameter group.

*) The compiler is programmed entirely in ALGOL and requires a rather large area in the main storage for tables and variables.

parameters concerning source text:

sysdok
section
version
ldtext
init

parameters concerning description files:

descripfile
ldfile
section

parameters concerning listing:

list
listout
paper
xref

parameters concerning compilation:

names
vardecl
size

parameters concerning test output

test
testout

fig. 4.2: Survey of fp-parameter key-words

Parameters concerning source text:

Sysdok file

The source text for the LD description can be read either from a Sysdok file or from an ordinary text file. When read from a Sysdok file the following parameter groups should be mentioned:

```
sysdok.<sysdokreg_name>
section.<section_number>
version.<version_number>
```

The Sysdok parameter states which Sysdok file the text is to be read from.

Standard: sysdok.sysdokfile

Section

The section parameter states which section in the Sysdok file the text is to be read from. The section number is given in the usual Sysdokmanner with the main section and subsection numbers, if any, separated by decimal points. The parameter cannot be left out, if the text is read from a Sysdok file.

Version

The version parameter states the version of the Sysdok section to be translated.

Standard: last version

Textfile

When reading from an ordinary text file, the name of the text area, from where the LD description is to be read, is stated with:

```
ldtext.<ldtext_names>
```

Standard: if this parameter is left out, the compiler reads from the Sysdok file. If the parameter is stated, no Sysdok parameters must then be stated.

init.<initials>

Initials

This parameter group states the initials of the person responsible for the activation of the LD compiler, to be printed in the log. <initials> is a string of max five letters. Standard value (= empty textstring) should not be used.

Parameters concerning description files:

DB descr.
file

The name (or names) of the description file(s) containing the actual DB description(s) can be specified by the following parameter group:

<pre>descripfile {.<descripfile_name>}₁³</pre>
--

In a normal LD description only one descripfile name is used, but in an LD description containing 'record output' or 'record input' up to 3 descripfiles can be specified. In this case, the first name corresponds to db_file_index = 1 (cf. section 3.4), the second name to db_file_index = 2, and so on.

Standard: descripfile.descripfile

LD file

The LD file produced by the compiler is treated as specified by the following command unit:

<pre>ldfile. { no yes <descripfile_name>} }</pre>

'ldfile.no' means that no binary LD file is generated. A descripfile name indicates the name of the descripfile, in which the compiled LD description will be stored, if it is formally correct. This file must be one of the descripfiles stated for the DB description.

'ldfile.yes' means the same as standard: the compiled LD description is stored in the first-mentioned DB description file.

Section

The binary LD file is always stored in the descripfile section with the same section number as the source text in the Sysdok file. If, however, the source text is read from a text file, a section number for the descripfile must be specified by

```
section.<section_number>
```

where <section_number> is given in the usual Sysdokmanner.

Parameters concerning listing:

```
list. { yes
      { source
      { no }
```

```
xref. { alpha
      { num
      { no }
```

```
listout.<listout_name>
```

```
paper. { <boss_paper_format_code>
      { <lines_pr_page>.<characters_pr_line> }
```

List

The list command states whether an edited listing of the LD description is produced or not.

'list.source' indicates that a listing is to be produced without the compiler-generated comments in the record input/record output section.

Standard: list.no

Xref

The xref command indicates the wanted format of the variable cross reference list, which can be produced only if list.yes or list.source is stated (cf. section 4.2).

xref.alpha: The xref list is arranged alphabetically according to the variable names. Furthermore, a simple variable number list is produced, if the variables are declared with numbers.

xref.num: The xref list is arranged numerically according to variable numbers. This is of meaning only if the variables are declared with numbers. In this case the simple variable number list is suppressed.

xref.no: The listing is produced with no cross reference lists.

Standard: xref.alpha

Listout

The listout parameter states the name of the disc-area on which the listing is written. If the area does not already exist, it will be created as a temporary file, which is automatically converted on the local printer. If the area already exists, the user must provide for the converting.

Standard: listout.listout

Paper

The paper command indicates the format of the edited listing by one or two integers.

<lines_pr_page>.<characters_pr_line> is stated if the user wishes an individually selected maximal number of lines printed on a page. If <lines pr page> exceeds 15 the value will define this maximum. Otherwise it indicates no automatic line-number-controlled new-page operation during the listing, that is, the listing will be more compact.

The value of <characters_pr_line> has no effect in the present implementation, but its presence with the preceding point indicates the specific meaning of the 'paper' command unit.

<boss_paper_format_code> indicates a standard format for a Boss paper type, i.e.

0 = monitor paper (62 lines/page)

2 = A4-horizontal (40 lines/page)

Standard:

When reading from a Sysdok file: as stated in the owner information.

When reading from a textfile: paper.0.

Parameters concerning compilation:

Language code

The language code for selection of alternative variable names (cf. section 3.1.1) can be specified by the following parameter group

names.<language_code>

Standard: names.0 means that the ordinary names are used.

Variable
declaration

If the programming language specified in the head line of the LD description is ALGOL, or if it is DUET and some of the variables are declared with an asterisk (cf. section 3.1.1), the compiler produces a text file of variable declarations to be incorporated in the application program or in the DUET control program respectively.

This declaration file can be specified by the parameter group

vardecl. { no
<vardecl_name> }

'vardecl.no' means that no declaration file is generated.

<vardecl_name> indicates the name of the declaration file. If this file does not already exist, it will be created as a temporary file.

Standard: vardecl. vardecl

Size

The size of the internal compiler tables can be changed by one of the following two parameter groups:

```
size.<extension_percent>
size.minus.<reduction_percent>
```

size.<extension_percent> states in percentages the expansion of all the compiler's internal tables. It should only be used, if the compilation terminates with an index-error, and the maintenance group must be notified, when this becomes necessary. Size.100 means doubling of all tables.

Size.minus.<reduction_percent> states a reduction of all the compiler's internal tables, which can be usefull when compiling a relatively small LD description in a limited care. Size.minus.50 means halving of all tables.

Standard: size.0.

Utilization
survey

The log produces by the compiler (see 4.2) contains a complete list of all the arrays that can be expanded, with a specification of how much was used in the current activation. In this way the user has a chance to monitor if the use of some arrays are approaching the limit, and possibly to report the expected overflow in due time to the maintenance so that the next version of the compiler will contain arrays which are large enough.

Note! The emergency expansion will influence the size of all the expandable arrays, independent of whether it is necessary or not. This will inevitably cause the compiler to require a process size that is considerable larger than necessary. Therefore, report all new recognised array overflows to the maintenance.

Parameters concerning test output:

Test

These parameters must only be used in agreement with the maintenance group when an error in the Sodald compiler occurs.

$\left\{ \begin{array}{l} \text{testa} \\ \text{testb} \\ \text{testc} \\ \text{testd} \\ \text{teste} \\ \text{testf} \\ \text{testg} \\ \text{testh} \end{array} \right\}$	$\left\{ \begin{array}{l} \text{.yes} \\ \text{.no} \\ \left\{ \text{.<number>} \right\}_1^{24} \\ \text{.not} \left\{ \text{.<number>} \right\}_1^{24} \end{array} \right\}$
--	---

Indicates which testbits are to be inserted in the respective test variables. Testbits are numbered from 0 to 23.

Standard: no textbits

$\text{testout.<testout_name> } \left\{ \text{.extend} \right\}_0^1$

Testout

Indicates the name of the area, where the test output is printed. If 'extend' is stated, the area is extended, if necessary, to contain the test output. If not, the test output is written cyclically in the area, in which case the testout area must contain at least 5 segments. The testout area is neither created nor converted automatically.

Standard: testout.testout

```
sodalld  init.pl  ldtext.lddescript  dbfile.descrfile  ldfile.no,
          list.yes  listout.ldlisting,
          paper.2  vardecl.no  size.50
```

Translation of an LD description in the text file 'lddescript' using a DB description file in 'descrfile' and with no LD file produced. A listing is required on the area 'ldlisting' to appear edited on standard format A.4 horizontal. A text containing declarations for an ALGOL program/DUET system shall not be produced, and the translation should be performed with a 50% expansion of the internal tables

```
sodalld  init.eah  sysdok.systest35  section.3.34.5.27.8  version.13,
          dbfile.descrfile  list.yes  names.2
```

Translation of an LD description present as version 13 in the sysdok file 'systext35' from the specified section number. The DB description will be read from and the LD file generated in the file 'descrfile'. A listing will be produced on the standard area 'listud' edited in standard format 'monitor'. A possible text with declarations for an ALGOL program/DUET system will be generated on the standard file 'vardecl'. Alternative operational names will be generated according to language code 2.

Fig. 4.3. Two examples of activation commands for the LD compiler

4.2 Listing and log

Listing

The LD description text will be listed by the compiler if required (see 4.1). The listing is edited by indentions, formatting, page headings, and generated comments to improve the readability and present the text in a pleasant way. However, the editing is completely loyal to the original text, so that it appears in the listing as a correct subset except for extra spaces.

New page

The listing consists of a number of pages each one headed by a compiler generated text. A new page will appear at the beginning, at the head of the value spectra table and at the head of each new set declaration. Furthermore new pages will appear whenever the standard number of lines pr page (62) have been printed, unless this number has been changed by the 'paper' command unit in the activation (see 4.1).

Page heading

The generated heading contains the following fields.

1. line: - the constant text 'SODA-LD'
 - the LD identification from the head line of the text (not on page 1)
 - the time of the compiler activation in the form:

dd.mm.19yy - hh.mm

2. line: - the name of the area containing the LD text or in SYSDOK mode: the section number
3. line: - the standard date identification of the text file (from the catalog) or in SYSDOK mode: the version number.
- the page number

Line numbers

Each line appearing in the input text will appear in the listing preceded by two line numbers:

- the external number which for ordinary text files is equivalent to the BOSS line number and for SYSDOK files to the SYSDOK line number
- the internal number which is a sequence number independent of paging and external numbers. The internal number is referred in the error messages

Vertical editing

The compiler will insert an extra blank line (without line numbers) before lines starting with the reserved words:

- 'norm values'
- 'record sets'
- 'usage'
- 'log'
- 'identspec'
- 'dspec'
- 'mspec'
- 'fieldspec'

Horizontal
editing

The compiler may edit the lines in several ways:

- indentation: Insertion of 2 or 4 extra spaces before the line as a simple representation of the structure in the description.
- tabulation: For detail lines of the same kind the compiler will try to let the same type of fields appear in the same position under each other, In the variable table the use of variable names longer than 23 characters will cause the remaining fields in the respective lines to appear out of position.
- documentation: In lines with field/variable associations the compiler will generate a text repeating the association but with use of the corresponding names derived from the variable declarations and the DB description. These generated comments are preceded by the special symbol -!- to indicate the start of a text not present in the input.

Error messages

Errors recognized during the reading of the text will appear in the listing at the correct location. The erroneous line may be printed without a possible editing. Errors recognized later on during the compilation will be printed after the listing with reference to the internal line numbers.

Cross refer-
ence list

After the listing of the LD description the compiler produces cross reference lists of variables, value spectra, norm values, and sets.

Variable
references

The variable reference list can be printed in alphabetical order according to the variable names or in numerical order according to the variable numbers, the latter of course only if variables are declared with explicit numbers.

The variable reference list - of which an example is shown in figure 4.4 - contains for each variable its name, number and type. For an explicit variable are given the w- and n-numbers, whereas for an associated variable or an implicit variable the vref-number for all fields belonging to this variable are shown. If more than one DB-file is used in the DB-description, the descripfile-index is shown in a parenthesis.

The definition line number is only printed if the variable is declared explicitly or associated.

The variable references are printed separately for each set. If the variable is used for record input or record output, the list contains the number of the logical file and the record types to which the variable belongs.

If the variable is used in a normal set definition, the reference line numbers are printed together with a possible association symbol. The reference line numbers are also printed if the variable is referred to in the value spectra table or the norm value table.

Variable
number list

If the variable reference list is printed in alphabetical order, and the variables are declared with numbers, the compiler will produce a variable number list in numerical order, containing the name, type and definition line for each variable.

The variable reference list and the variable number list can be suppressed by an fp-parameter in the call of the compiler (cf. section 4.1).

VARNAME	VNO	VARTYPE	VALUE-NORM DEF.	SET LOG.F.
NAME	V5	TEXT.40	W1	L5 S 3: *<=>24
NO	V3	WORD.	VREF1(1)	L4 S 3: < 23
OTHER←AMT	V24	LONG.2	VREF8(2)	IN: R12 I1202 OUT: R20 I1402
RC3600	V33	WORD		OUT: R20 I1403
RC4000	V31	WORD		OUT: R20 I1403
RC6000	V32	WORD		OUT: R20 I1403
REC←NO←CF	V14	WORD		OUT: R20 I1403
RECLENGTH	V13	WORD		IN: R12 I1202 OUT: R20 I1403
RECSTATE	V17	WORD		OUT: R20 I1403
RECTYPE	V15	WORD		IN: R12 I1202 OUT: R20 I1403
SUBST←AMT	V23	LONG.2	VREF8(2)	IN: R12 I1202 OUT: R20 I1402
TYPE	V12	WORD		OUT: R20 I1403 S 3: = 21

fig. 4.4: Example of a page of the variable reference list.

Log

The compiler will document the result of the activation in a log printed on standard output. The log contains the following fields:

Head:

- the constant text: 'SODA-LD LOG'
- the LD identification from the head line of the LD description
- the time of the compiler activation dd.mm.yy - hh.mm

Binary files:

- the area name of the DB description file
- the following description of the LD file: *)
 - * a version number. This is increased by one every time the LD file is replaced by a new version. In SYSDOK mode it is the version number from input.
 - * the initials as supplied by the activation command unit 'init' (see 4.1)
 - * the activation date and time
 - * the LD identification from the head line of the LD description

Text files:

- the following description of the LD text file:
 - * the area name of the file
 - * the standard date identification from the catalog or in the SYSDOK case: the version number

*) This information is represented in the header record of the LD file and serves as its identification. It is reproduced in the log from the DUET system (cf. ref. 5).

- the following description of the possible
declaration file:

- * the area name of the file
- * the activation date and time

Survey of table utilization:

This is a list of the various arrays of the compiler in which the internal representation of the description is stored. For each array the list specifies the declared number of elements and how many of these were used during the compilation. The survey will specify a possible emergency expansion of the arrays (cf. 4.1 the 'size' command unit).

4.3

Reaction on errors

The compiler may recognize two different kinds of errors:

- formal errors in the activation parameters
- formal errors in the LD description

Activation
errors

Formal errors in the activation parameters will be documented on standard output. Syntactical errors are indicated by the following symbols:

<*> if a new command unit appears as an incorrect termination of the previous unit

<*> if a wrong key word or parameter is recognized

Consistency errors or illegal values of parameters are specified by messages of the following type:

*** error in activation command:

error specification text

The recognition of any error in the activation parameters will result in immediate termination of activation.

LD errors

Formal errors recognized during the compilation of the LD description will be documented on standard output and in the LD listing if such one is produced.

The error messages on standard output have the format as shown in the following example:

Std. output
messages

LINE NO	CHAR NO	SYNT. UNIT	ERROR
832	15	s65	12: ILLEGAL SET NUMBER 65
1387	0		81: TYPE CONFLICT 3 7

The line number refers to the internal numbers in the listing.

The char number refers to the number inside the current line of the last character read when the error was recognized. This value is relevant only for errors recognized during the input phase of the compiler.

The column denoted 'SYNT.UNIT' shows the last read syntactical unit.

The error specification consists of an error number, an error text and possibly some specifying error parameters. The full list of error specifications with explanations is given in appendix B.

Messages in
the listing

Messages about errors recognized during the input phase will appear in the listing at the location where they are recognized. They have a format as the following example:

```
***** 12: ILLEGAL SET NUMBER 65
```

The message is followed by the erroneous line preceded by **.

Errors recognized in the later phases of the compiler will appear after the listing. They have a format as the following example:

```
***** line 1387 81: TYPE CONFLICT 3 7
```

Check for errors

An error which causes termination of the compilation will set the "ok bit" to 'false'. An error in the LD description involves that no binary LD file is updated, and the "warning bit" is set to 'true'. These error bits can be checked for possible termination of the job in case of errors:

```
sodald ..... ; compilation
if warning.no ; terminate the job
if ok.no ; if any errors have
    finis ; been discovered
..... ; else continue job
```


5. The SODA DBMS

This section describes the seven SODA operations that can be activated from a program in order to access a data base. The reader is assumed to be familiar with the notions and principles of SODA as described in section 2 and with the concepts and semantics of the SODA LD language as described in section 3.

The use of the DBMS operations is checked for legality in several ways:

Static check

Any access operation on a given set requires that the name of the operation is specified in the usage entry of the set declaration.

Although much has been done to obtain a uniform set of rules, deficiencies in the file systems or a wish to avoid too much overhead 'behind the back' have caused some restrictions in what is permitted for sets associated with list files. The LD compiler will check the usage entry according to the following scheme: (+ allowed, - not allowed)

	singular sets type L	subscripted sets
GET	+	-
NEXT	-	+
LOOKUP	-	-
PUT	+	+
CREATE	-	+
DELETE	-	+
NEWSET	-	+
CLOSEFILE	+	+

Dynamic check

Most of the operations are only permitted for certain values of record state or if the set is open for sequential access. This is checked by the DBMS and an error result will occur. The rules and the possible results that occur if they are violated are specified individually in the descriptions of the operations.

Furthermore the legality of the operations may depend on the structure of the DB itself and of its actual contents when the operations are activated. Also the state of associated sets (mother/daughter relations) may influence the legality. These rules and the results are as well specified individually in the descriptions.

A special rule
for NEWSSET

In the present implementation a special rule applies for subscribed sets: One set only, among those associated with the same file, can be open for sequential access at the same time. Every activation of NEWSSET on a subscribed set will automatically turn any other open set associated with the same file into a closed state.

Result:

The result of an operation is communicated back to the application program in a way depending on the programming language:

In DUET:
result

The single integer result-value specified in the individual descriptions of the operations is available for the program in the possible variable declared as result.soda (cf. 3.1.1).

- recno The system generated ident value (recno key) provided on certain operations on set type B or L is available in the possible variable declared as result.recno (cf. 3.1.1).
- errors When a normal result occurs the DUET system will activate an automatic error reaction of one of the three kinds: data error, program error, or system error. The DUET manual (ref. 5) contains a comprehensive description of the error handling mechanisms in DUET.
- In ALGOL:
result The result value is available as the value of the procedure call.
- recno The system generated ident value (recno key) provided on certain operations on set type B or L is available in the standard variable 'recno_cf'.
- errors Information about whether an error has occurred or not is available as the result value. However, by proper initialization of a global variable the SODA DBMS may activate a user coded error procedure in which any error action could be performed. In order to simplify the coding of this procedure the result values are defined so that the same error situation will cause the same result value to be assigned for all SODA operations. In 6.4 and 6.5 the possible communication between SODA and an error procedure is described.

General frame
of descrip-
tions

The individual descriptions of the SODA operations in the remaining part of section 5 are all structured in the following sequence:

- 1: activation format
- 2: specification of main rules for dynamic legality
- 3: macro algorithmic specification of actions performed during activation. Each step is executed in the specified order. If an error is recognized the remaining steps are skipped and an error specific result value is returned
- 4: specification of relevant result values and possible changes in record state and sequential state or other effects caused by the activation. The result value column supplies two values:

- the value returned in the result variable and in ALGOL as value of the procedure
- the possible DUET error type and number (cf. ref. 5):

D: data error
P: program error
S: system error

5.1 GET

Activation: get (setno)

Legality: No restrictions

The purpose of GET is to provide from the set in the DB, a record identified according to the ident specification and establish it as the current record of the set.

GET operates in the following manner:

- check for legal setno and usage
- registration of the ident values according to the ident specification
- activation of the log procedure if 'log before' is specified for the current set and operation
- physical DB access with possible reaction: 'No record with that key'
- check for record type membership of set
- check for record membership of set according to a possible restriction
- check for correct record length
- activation of the log procedure if 'log after' is specified for the set
- transfer of field values to variables according to possible field specifications
- for settype L: check that the record is connected to mother records according to the mother specification and transfer of specified m-ref values to variables.

Result		Situation	New record state
0	-	Record read from DB	db currec
1	D11	Record not in the set	empty
2	P9	Illegal setno	"
3	P10	Usage GET not specified	"
5	P12	Spill during transfer of ident values or index during transfer of values to variables	"
17	S7	For set type B: checksum error	"
18	S8	Incorrect record length	"
19	S9	Record not correctly connected to mother records	"
24	S6	Internal error (please contact main- tenance)	"
25	S16	Internal error (please contact main- tenance)	"
27	D15	File not open (data entry)	"

Set sequential state is not affected by activation of GET.

5.2 NEXT

Activation: next (setno)

Legality: record state: No restrictions
 sequential state: Open

The purpose of NEXT is to provide from the DB the next record in the set relative to set sequential position and establish it as the current record of the set.

NEXT operates in the following manner:

- check for legal setno and usage
 - check for sequential state
 - for subscripted sets: check that the last activation of NEWSET for sets associated with the current list file, was for the current set. Furthermore, for first NEXT after NEWSET: check that the mother record has not been removed from the DB
 - activation of the log procedure if 'log before' is specified for the set.
 - repeated physical access of records in the associated file. For subscripted sets access is performed in the chain associated with the mother set. A possible reaction of this may be: 'No more records in the set'.
- For each access:
- check for record type membership of set
 - check for record membership according to a possible restriction
 - check for correct record length

- activation of the log procedure if 'log after' is specified for the set
- transfer of field values to variables according to possible field specifications
- for subscripted sets: check that the record is connected to mother records according to the mother specification and transfer of specified m-ref values to the variables

Result	Situation	New record state	New seq. state	
0	-	Record read from DB	db currec	open
1	-	No more records in the set	empty	closed
2	P9	Illegal setno	"	"
3	P10	Usage NEXT not specified	unchanged	unchanged
6	P13	Set closed for seq. access by NEWSET on another set	empty	closed
11	P18	Mother record removed since NEWSET	"	unchanged
15	P25	Illegal BS.operation, position after eof	"	closed
17	S7	Checksum error (for set type B)	"	unchanged
18	S8	Incorrect record length	"	"
19	S9	Record not correctly connected to mother records	"	"
24	S6	Internal error (please contact maintenance)	"	"
25	S16	Internal error (please contact maintenance)	"	"
26	P27	Set closed for sequential access	"	closed
27	D15	File not open (data entry)	"	"

5.3 LOOKUP

Activation: lookup (setno)

Legality: no restrictions

The purpose of LOOKUP is to check in a simple and relatively fast way whether a record defined according to the ident specification is present in the DB as a member of the set.

LOOKUP operates in the following manner:

- check for legal setno and usage
- registration of the ident values according to the ident specification
- activation of the log procedure if 'log before' is specified for the set
- physical DB access with the possible reaction: 'No record with that key'
- check for record type membership of the set
- check for record membership of the set according to a possible restriction

Result		Situation
0	-	Record is a member of the set
1	-	Record is not a member of the set
2	P9	Illegal setno
3	P10	Usage LOOKUP not specified
5	P12	Spill during transfer of ident values
25	S16	Internal error (please contact maintenance)
27	D15	File not open (data entry)

Record state and set sequential state are not affected by activation of LOOKUP.

In the present implementation LOOKUP can only be used on sets of the settype M.

5.4 PUT

Activation: put (setno)

Legality: record state: $\left\{ \begin{array}{l} \text{db currec} \\ \text{new currec} \end{array} \right.$

sequential state:

subscripted sets: Open

Other sets : no restrictions

The purpose of PUT is to transfer the current record of a set to the DB. If the current record was established by CREATE it will be inserted in the DB as a new record and if it was established by GET or NEXT it will replace its former instance in the DB.

PUT operates in the following manner:

- check for legal setno and usage
- check for legal record state
- for subscripted sets: check for sequential state and that the last activation of NEWSET was for the current set.
- activation of the log procedure if 'log before' is specified for the set

Then the operations depend on the record state:

db currec (after GET or NEXT)

- if the current record of the set is not the current record of the file (due to operations on other sets associated with the same file) then reestablish it by the necessary physical file operations
- transfer of values from variables to relevant fields according to the field specifications

- activation of the log procedure if 'log before' is specified for the set
- for settype B: generation of new record checksum
- physical transfer of the record back to the DB
- activation of the log procedure if 'log after' is specified for the set

new currec (after CREATE)

- establishment of a physical record which is initialized with zeroes or empty strings
- transfer of values from variables to relevant fields according to the field specification. Hereby possible repeating groups are fixed concerning the number of repetitions, and their control fields are assigned
- assign of record length and record type (as defined by CREATE)
- for set type B generation of the record checksum
- activation of the log procedure if 'log before' is specified for the set
- insertion of the record into the DB depending on the set type:

M: according to ident values as recorded at the time of the previous CREATE

B: after the last record in the physical file

L: as the next record in the chain connected to the mother record of the subscripted set. *)

- for subscripted sets:
 - transfer of ident values for secondary mother records to which the current one should be connected according to possible mother specifications

*) The user familiar with the CF system will know that the insertion of a new record after recognizing 'end chain' where the last record was deleted, is not permitted. SODA circumvenes this restriction by a 'behind-the-back' reaccessing of the records in the chain from mother record to 'end chain' before the new record is inserted.

- physical access of and connection to these records 'next to mother' according to the DB description
- activation of the log procedure if 'log after' is specified for the set

Result	Record state	Situation	New record state
0	- DB currec	Record transferred back to DB	empty
0	- new currec	Record inserted in DB	"
2	P9 ~	Illegal setno	"
3	P10 ~	Usage PUT not specified	unchanged
4	P11 empty	PUT not allowed in empty record state	empty
5	P12 ~	Spill or index during transfer of field values	empty
5	P12 new currec	Number of repetitions for repeating group exceeds maximum	"
6	P13 ~	Set closed for sequential access by newset on another set since current record was established	"
8	D12 new currec	For set type M: Record is already in DB	"
9	P16 DB currec	Current record removed from DB via other set	"
10	P17 new currec	For subscripted sets: Secondary mother record is not present in DB	"
11	P18 "	For subscripted sets, first insertion: Mother record removed from DB since NEWSET	"
21	S10 "	For set type M and L: File cannot be extended	"
22	S11 "	For set type M and L: Record cannot be inserted for a reasonable price, or length error x) (see ref. 1)	"
24	S6 ~	Internal error (Please contact maintenance)	"
25	S16 ~	Internal error (Please contact maintenance)	"
27	D15 ~	File not open (data entry)	"

Set sequential state is not affected by PUT.

x) A length error can only occur in case of an error in the physical file, or in the combination of versions of the physical file and the sodald file.

5.5 CREATE

Activation: create (setno, record_type)

Legality: record state: no restrictions

sequential state:

for subscribed sets: Open

else : no restrictions

The purpose of CREATE is to provide from the program a record of the specified type and establish it as the current record of the set. On a following PUT it will be inserted in the DB.

CREATE operates in the following manner:

- check for legal setno and usage
- for subscribed sets: check for sequential state and that the last activation of NEWSET concerning this physical file was for the current set
- check that the specified record type is declared as a member of the set
- for set type M: Registration of ident values according to the ident specification
- for subscribed sets: Registration of ident values for possible secondary mother records according to mother specifications.
- activation of the log procedure if 'log before' is specified for the set
- possible assign of standard values to variables according to field specifications
- activation of the log procedure if 'log after' is specified for the set

Result		Situation	New record state
0	-	Current record created from program	new currec
2	P9	Illegal setno	empty
3	P10	Usage CREATE not specified	unchanged
5	P12	Index during transfer of standard values	new currec
6	P13	For subscripted sets: Set closed for sequential access by NEWSET on another set	empty
14	P21	Record type not member of set	"
24	S6	Internal error (Please contact maintenance)	"
25	S16	Internal error (Please contact maintenance)	"
26	P27	For subscripted sets: The set is closed for sequential access	"
27	D15	File not open (data entry)	"

Set sequential state is not affected by CREATE.

5.6 DELETE

Activation: delete (setno)

Legality: record state = $\begin{cases} \text{new currec} \\ \text{DB currec} \end{cases}$

sequential state:

for subscripted sets: Open

else : no restrictions

The record to be deleted must not have any daughter records in the DB.

The purpose of DELETE is to remove from the DB the record which is current of the set.

DELETE operates in the following manner:

- check for legal setno and usage
- for subscripted sets: check for sequential state and that the last activation of NEWSSET was for current set

The remaining actions are carried out only if record state = DB currec.

- for set type B: check the position of current record; all other sets associated with the same file having a current record after this position, will be closed.
- for set type M and L: possible physical re-access of the current record if other sets have operated last on the file

- for settype M: check that the record has no daughter records
- activation of the log procedure if 'log before' is specified for the set
- physical removal of the record from the DB
- activation of the log procedure if 'log after' is specified for the set

Result		Situation	New record state
0	-	Current record deleted	empty
2	P9	Illegal setno	empty
3	P10	Usage DELETE not specified	unchanged
4	P11	DELETE not allowed in empty record state	empty
6	P13	Set closed for sequential access by NEWSSET on another set	empty
7	P14	Daughter records associated with current record	unchanged
9	P16	Current record removed from DB via other set	empty
23	S12	For settype M: Last record in file must not be deleted	unchanged
24	S6	Internal error: (Please contact maintenance)	empty
27	D15	File not open (data entry)	empty

Set sequential state is not affected by DELETE.

5.7 NEWSSET

Activation: newset (setno)

Legality: for subscripted sets: record state for the mother set
must be: DB currec

The purpose of NEWSSET is to open a set for sequential access. For singular sets set sequential position is initiated according to a possible ident specification. For subscripted sets the scan is initiated to comprise the daughter records of the current record in the mother set.

NEWSSET operates in the following manner:

- check for legal setno and usage
- for set type M and B: transfer of ident values according to a possible ident specification. No ident specification implies that all ident values are zero. The ident values define set sequential position to be just before a possible record with that key
- for set type B: check that the derived set sequential position is not after the last record of the file
- for subscripted sets:
 - check for record state of the mother set
 - check that the record type of the current record in the mother set may have the current set as daughter set
 - registration of the ident values of the mother record
- activation of the log procedure if 'log before' is specified for the set
- for set type B: positioning to set sequential position in the the file
- activation of the log procedure if 'log after' is specified for the set

Result		Situation	New sequential state
0	-	Set open for sequential access. Position initialized	open
2	P9	Illegal setno	
12	P19	Illegal record state in mother set	closed
13	P20	Illegal mother record type	closed
15	P25	For set type B: Initial position after last record in file	closed
24	S6	Internal error (Please contact maintenance)	closed
27	D15	File not open (data entry)	closed

Record state will be 'empty' regardless of the result.

For subscribed sets: The activation of NEWSSET will imply that other subscribed sets associated with the same file will have their sequential state changed to 'closed'. *)

*) This is so due to a design error in the present implementation. A change is planned that will restrict the side effect to concern sets only which are associated with the same chain (list entry in Database80).

5.8 CLOSE FILE.

Activation: close_file (setno)

Legality: no restrictions

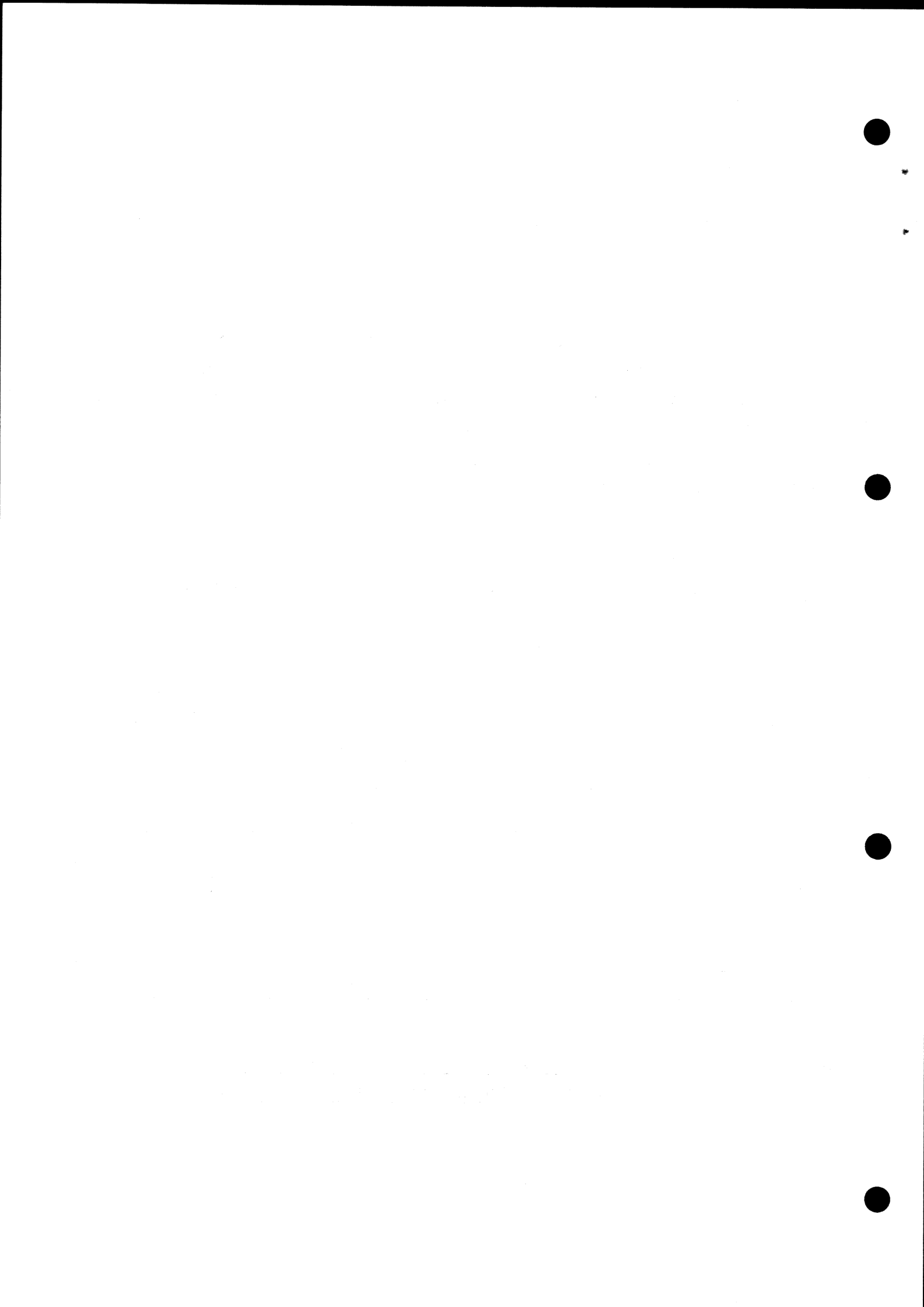
The purpose of CLOSE_FILE is to finish the physical access on the file of the current set. Usually all physical accesses are performed by the DBMS, but in some special cases (f.ex. before a sorting) the user may want to control this process.

CLOSE_FILE operates in the following manner:

- check for legal setno
- finish the last physical acces
- assign record state = closed for all sets connected to the same file.

Result		Situation	New record state	New seg. state
0	-	Physical access finished	empty	closed
2	P9	Illegal setno	unchanged	unchanged

Obs! The record state and sequential state will be empty/closed for all sets connected to the same physical file.



6. How to include the SODA DBMS in a program

This section describes the elements of the SODA DBMS and how it is incorporated in a program. DUET application programmers may stop the reading here but designers and programmers of DUET systems and ALGOL application programmers will need the information presented in this section.

6.1 ALGOL block structure and SODA program texts

A DUET system or an ALGOL application program employing SODA must be structured in two block levels as shown in fig. 6.1, or at least contain this structure at the same inner block level *). We shall refer to this structure as 'the program'.

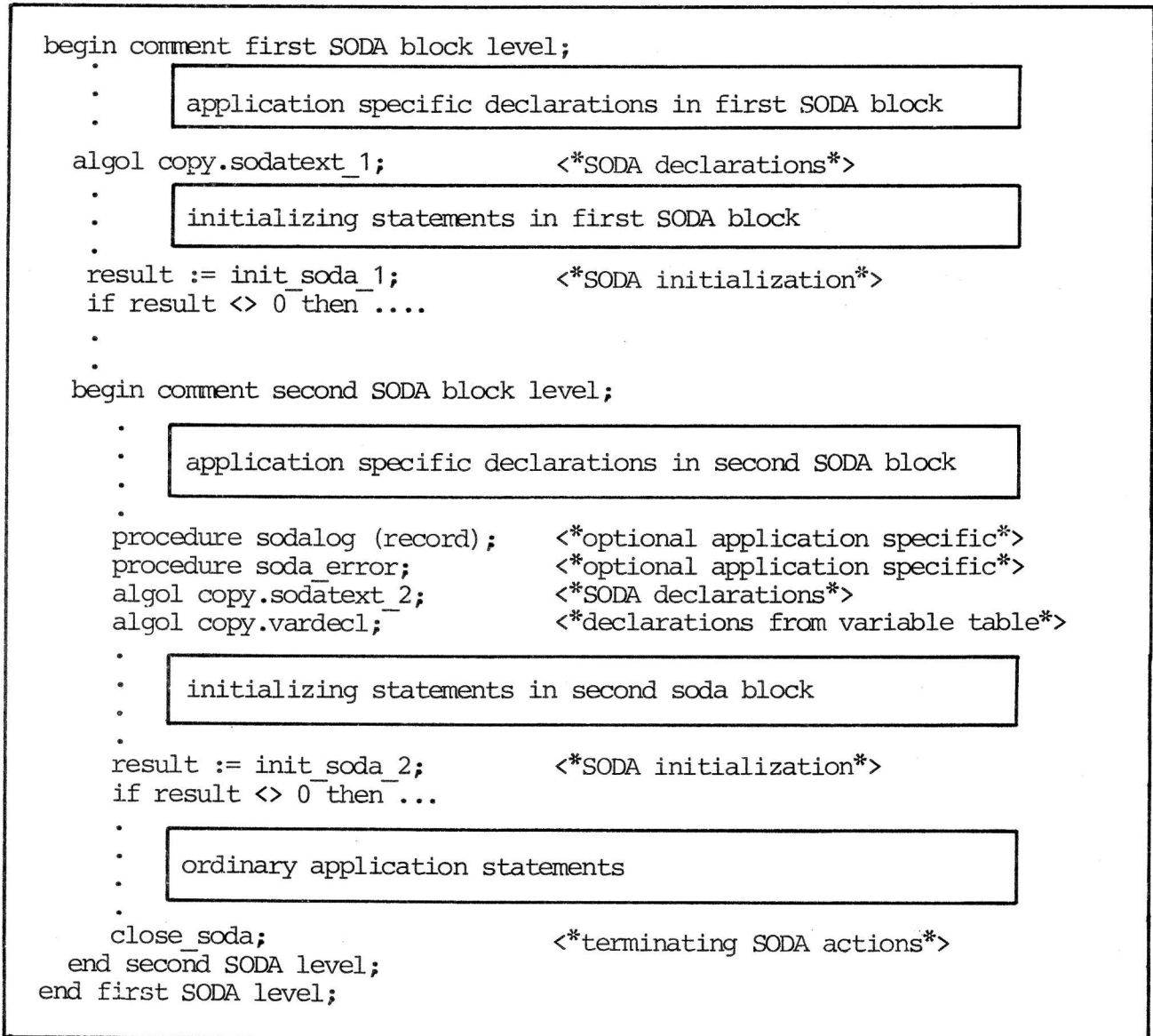


Fig 6.1. Block structure of a program employing SODA

*) Designers of DUET systems are advised to consult ref. 5 section 5.1 with the corresponding fig. 5.2 which contains more DUET specific details.

The SODA DBMS is programmed in ALGOL as a set of procedures and variable declarations which must be incorporated in the program before compilation by means of 'algol copy'. The SODA program text is available in four text files organized as shown in fig. 6.2.

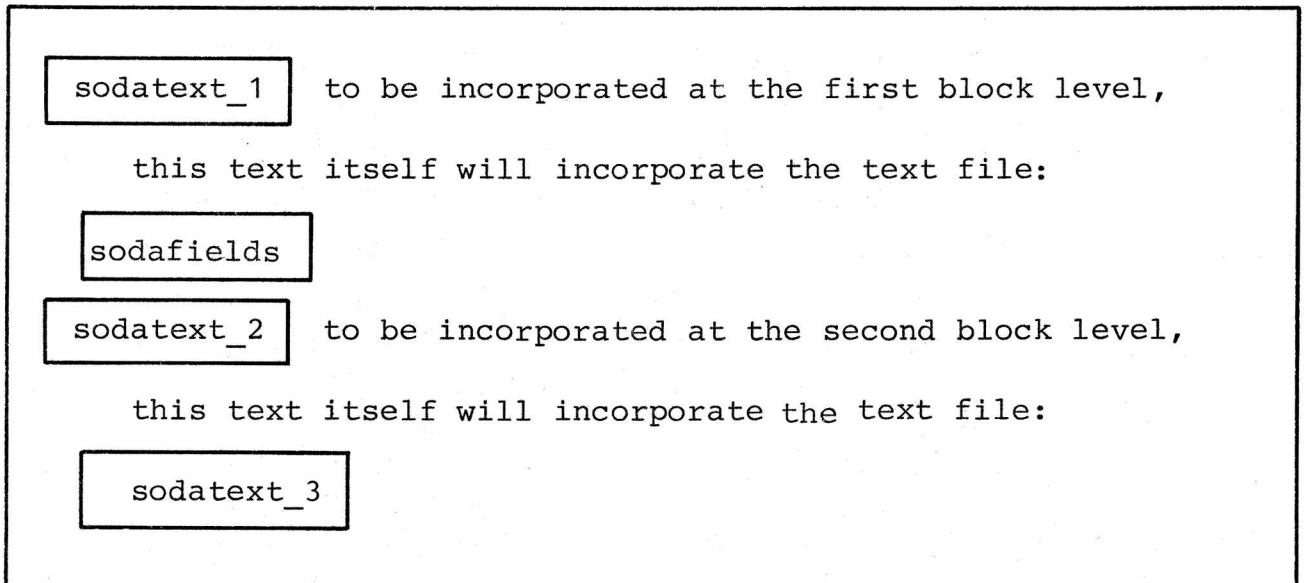


Fig. 6.2. The organization of the four SODA text files.

In all normal situations the user can regard sodatext_1 and sodatext_2 independent of this internal copy structure.

Returning to fig 6.1 the following further explanation may be useful:

At the first block level 'sodatext_1' must be incorporated among the possible declarations before the first proper statement of the block. 'sodatext_1' contains certain variable declarations and especially the procedure 'init_soda_1' which must be called once at this block level after some variable initializations as described in 6.2.

At the second block level the user may declare the two application dependent procedures 'sodalog' and 'soda_error' as explained in 6.4 and 6.5 respectively. Furthermore among the declarations on the second block level, before any statements, the declarations generated by the LD compiler in the file 'vardecl' must be incorporated. For a DUET system this incorporation is optional.

Also on this block level, before any statements 'sodatext_2' must be incorporated. It contains further declarations of variables used by the DBMS and declarations of most of the fixed size and all variable size internal tables. Their length are determined by the LD compiler and communicated via the LD file and the call of init_soda_1 to the program. Moreover 'sodatext_2' contains procedures for all the DBMS operations and the procedure 'init_soda_2' which must be called before any DBMS operations are activated as described in 6.2.

Finally 'sodatext_2' contains the procedure 'close_soda' which must be called at the end of execution before exit of the second block level.

The SODA procedures use some SLANG code procedures, which must be translated by calling

```
i prim
i sdtrans
i tduetcode
```

before calling the ALGOL compiler.

6.2 Initiating the SODA DBMS

As mentioned in 6.1 the first block level must contain a call of the procedure `init_soda_1` and the second block level a call of `init_soda_2`. This section contains a description of the primary function of these procedures and specifies the set of variables that is used for communication with them.

`init_soda_1`: This procedure performs the following tasks:

- reading of the header record of the LD file
- check for identification and version of the LD file
- assignment of upper limit values for variable size arrays to be declared in the second block level.
- assignment of other values from the header record for use in the surrounding program or by the DBMS at the second block level.

The physical file containing the LD file is accessed via a locally declared zone.

Information is communicated to and from '`init_soda_1`' through a set of variables also declared in '`sodatext_1`'.

Before activation of `init_soda_1` the following variables must be assigned:

<code>beskrivnavn</code>	real array (1:2) The area name of the LD file
<code>ld_afsnit_nummer</code>	long The section number that serves as key for the LD file in the physical file (cf. 4.1, LD file generation)

ld_ident integer
 The identifying number from the head line of the LD description (cf. fig. 3.1)

ld_version integer
 The version of the LD file. If zero is assigned the version check will be suppressed

sd_extend_buf integer
 The size measured in reals of the available space for buffer extension for each CF master file involved (cf. ref. 1)

Upon return the integer value of init_soda_1 will indicate the result as a bit pattern. A one-bit in the positions mentioned below will indicate an error while a zero-bit indicates OK.

bit 0 = 1 : No records with specified section number
 bit 1 = 1 : The specified section is not a SODA LD file
 bit 2 = 1 : The LD file is not the result of a correct compilation
 bit 3 = 1 : The version of the LD file does not match the specified version
 bit 4 = 1 : The header record of the LD file is missing
 bit 5 = 1 : The identification number of the LD file does not match the specified identification
 bit 23-6 : Not used (invariantly zero)

ld_brugernr integer
 The user number from the head line of the LD description (see 3)

ld_initialer	real array (1:1) The initials from the 'init' command unit in the activation of the LD compiler (see 4.1)
ld_navn	real array (1:3) The LD identifying name from the head line of the LD description (see 3)
ld_regdato	integer The compilation date as it appears in the log from the compiler
ld_regtid	integer The compilation time as it appears in the log from the compiler
ld_varsum	integer A compressed checksum of the variable numbers and names that for DUET systems will secure that the use of another version of the LD file than that present during compilation of the DUET program will not cause errors due to different variable declarations
sd_sprogkode	integer An internal representation of the programming language specification from the head of the LD description. 1: DUET 2: ALGOL
soda-giveup	integer Is initialised = 0. If this variable is changed before the call of 'init_soda2' to a bit mask containing '1 shift 1', then during the run, the DBMS will count the number of disc accesses for CF-files and BS-files in the variables 'soda_cfaccess' and 'soda_bsaccess', respectively.

init_soda_2: This procedure performs the following tasks:

- calculation and assign of base address variables
- reading of internal DBMS tables from the LD file
- initialization of the DBMS zone array according to physical file properties and opening of the zone elements.
- initialization of the CF 'chainref' array (see ref 1)
- initialization of the dynamic DBMS file and set tables

The physical file containing the LD file is accessed via a locally declared zone

The integer value of init_soda_2 will indicate the result of the activation in the following manner:

- 0: All tasks performed correctly
- 1-33: Error during reading of a DBMS table from the LD file
- 40: Unknown file (BS file only)
- 41: Blocklength of BS file does not match that derived from the DB description

The procedure does not require any special assign of variables before the activation and it will not communicate special values of interest for the program except those intended for the DBMS.

6.3 The log mechanism

As described in 3.2.4 the user may specify for each set whether a special user coded log procedure should be activated by the DBMS just before and/or just after physical access of the files associated with the set. The purpose of such an activation could be to monitor or survey certain operations on the DB in an application-specific way.

At the first SODA block level a dummy procedure 'sodalog' is declared in order to avoid any undeclared references from the DBMS procedures. At the second block level the user may re-declare 'sodalog' with the application-specific code. Then it will be this procedure that is activated from operations on sets where 'log' is specified.

The log procedure should be declared according to the skeleton shown in fig 6.3

```
procedure sodalog (record);  
  real array record;  
begin  
  .  
  .  
  .  
end;
```

Fig 6.3 Skeleton for declaration of a log procedure

When 'sodalog' is activated the parameter will contain an image of the record involved in the physical operation.*) This image is complete in all situations, except in the 'log before insert' situation for subscripted sets. Here the m-ref fields (located after the ordinary fields of the record) will have an undefined contents, because the necessary connect operations have not been executed yet.

In order to enable sensible operations the following DBMS variables and tables may be used, but not altered!

sd_sætnr integer
 The set number equal to that supplied as
 parameter to the DBMS operation

sd_procno integer
 Internal code defining the current DBMS
 operation:

- 1: GET
- 2: PUT in function 'update'
- 3: DELETE
- 4: CREATE
- 5: NEWSET
- 6: NEXT
- 7: LOOKUP
- 8: PUT in function 'insert'

*) No record is available in the following situations:
 'log before': NEWSET, CREATE, GET, NEXT, LOOKUP, PUT (UPDATE, INSERT)
 'log after' : NEWSET, CREATE, DELETE

In these situations the record parameter is an array in which the first element is zero (corresponding to the length field in a normal record).

sd_sætttype

integer array (1: max_setno)

A set table that may be subscripted with sd_sætnr.

It contains an internal code defining the set type and set kind:

- 0: undeclared set
- 1: set type M
- 2: set type B
- 3: set type L singular
- 4: set type L subscripted

filnr

integer

An internal number used as index in the zone array 'sdz' which is common for all the physical files. The name of the file or other relevant information can be derived from the zone descriptor of 'sdz(filnr)'.

Warning: The value of 'filnr' is a unique identification of the file in all runs performed with the same LD file. However, changes in the set section of the LD description may cause the internal file number to be altered for any number of sets

sd_log_before

boolean

true => log before

false => log after

sd_kædeindex

integer

This value is defined only when sd_procno = 8 and sd_sætttype = 4. It contains an index into the SODA table 'kædeinf', which holds the information necessary for connecting a listfile record to the mother records in a possible reestablishing of the database.

If the user needs more information than described here to design and program a log procedure the maintenance group must be contacted.

6.4 The DBMS error mechanism

The treatment of DBMS errors - that is, abnormal results from the operations as described in section 5 - is completely taken care of by the DUET system as described in ref. 5.

When the programming language is ALGOL the user may choose either to let the program check the result of each DBMS operation and take the proper action on errors, or to specify to the DBMS that it should activate a common error procedure before the operation returns with the abnormal result value.

The standard reaction is that SODA will not automatically activate such a procedure, but if the user in a statement after the call of `init_soda_1` assigns:

```
sd_alarm := true;
```

then the DBMS will activate the procedure:

```
soda_error
```

when abnormal results are recognized.

The text `soda_text_1` contains a declaration of 'soda_error' with a very rude and primitive action. It simply announces on standard output that some error has occurred on some set and then immediately terminates the execution. Surely, this is quite unacceptable for any real application program. The procedure is just a part of `sodatext_1` to avoid undeclared references. The intention, of course, is that the user supplies his own 'soda_error' on the second block level if 'soda_alarm' is set to true.

A user coded version of 'soda_error' would possibly document the situation properly and for each type of error it could then react in one of three ways:

- return to the DBMS, possibly after some attempts to cure the situation, so that the program can go on after return from the DBMS
- terminate the execution. In this case the procedure 'close_soda' should be activated before exit of the program
- jump to some location in the application program. Attempts after this to re-access the same set may for certain error types cause unpredictable results, especially for set type L and in case of sequential scans. However, in most cases a call of NEWSSET should reestablish a normal situation for the set.

From the error procedure the user has access to all the internal DBMS tables and variables. The following survey exposes those which may be of special interest for the error procedure. Users who look in vain in the survey for necessary information must contact the maintenance group.

sd_sætnr	}	see 6.3
sd_sættype		
filnr		

sd_procno	integer
	Internal code defining the current DBMS operation. NOTE a slight difference from the survey in 6.3

- 1: GET
- 2: PUT
- 3: DELETE
- 4: CREATE
- 5: NEWSSET
- 6: NEXT
- 7: LOOKUP

`sd_resultat` integer
The result value from the DBMS operation (see 5.1 - 5.7). It indicates the type of error

`sd_tilstand` integer
Indicates the record status and the set sequential status in the following way:

sd. tilstand	situation	record state	seg. state
0	after open	empty	closed
1	after get	DB currec	irrelevant
2	after put-direct	empty	irrelevant
3	after delete-direct	empty	irrelevant
4	after create	New currec	irrelevant
5	after newset	irrelevant	open
6	after next	DB currec	open
7	after put-sequential	empty	open
8	after delete-sequential	empty	open
9	after end-of-chain at delete	empty	open
10	after end-of-chain at next	empty	closed
11	as 10, but chain empty	empty	closed
12	after end-of-set at next	empty	closed

`sd_fejl` long array (0:20)
For some values of 'sd_resultat' a further specification of the error:

`sd_resultat = 18` illegal record length
`sd_fejl (1) = min rec.length from DB descr.`
`sd_fejl (2) = rec.length in record`

`sd_resultat = 5/25` error during field value transfers:
`sd_fejl (0) = number of recognized errors`
`sd_fejl (1:*)` error specification
The specification is packed with one element pr. recognized error. Bit 0-5 of each element contains an error code that determines the contents of the remaining element fields as explained below.

If any error with error code 1, 2 or 6 is recognized the value of `sd_result` will be 25, otherwise 5. Result 25 indicates a system error and should be reported to the maintenance group together with a printout of the contents of `'sd_fejl'`. The contents of `sd_fejl` is interpreted in the following way:

bit 0-5: error code:

- 1: Illegal entry address *)
- 2: Illegal instruction *)
- 3: Spill during transfer of values from variables
- 4: Illegal index for array or rep. group
- 5: Illegal number of repetitions specified for new rep. group
- 6: Illegal field address for rpg.

bit 6-17: rel.w.address for var. (err.code 3-5)

bit 18-19: base address code (err.code 3-5)

bit 20-23: number of var. bytes (err.code 3-5)

bit 24-47: depends on error code:

- 1/2: address of 'machine instruction'
- 3: 0
- 4: the index value
- 5: the specified number of repetitions
- 6: irrelevant

*) The transfer of field values is performed by the procedure `'sdtransfer'`. It operates like a virtual machine with a 'program' generated by the LD compiler from the field/variable associations.

6.5 Reserved ALGOL identifiers

Since the DBMS must be incorporated as ALGOL text in the program there is a chance for conflict between DBMS identifiers and user declared identifiers. To avoid this the user should consult the following list of reserved identifiers before starting the programming. Except for the variables and procedures mentioned in 6.3 and 6.4 the user is urgently advised not to interfere in any way with the reserved identifiers.

In general: All variables with prefix 'sd' or 'ld' are reserved.

At the first block level:

Variables: b_fællesadresse
 b_krit3
 k_afsnit_nummer
 k_afsnit_type
 k_afsnit_version
 k_fælles_adresse
 k_oversæt_tilstand
 max_iadr
 max_n
 max_var
 max_værdispec
 max_w

Procedures: init_ldfields
 læs_ldtab

At the second block level

Variables: array_base
 konst_base
 modernøgle_adr
 nøgle
 nøgleadr
 nøgleslut
 nøglestartadr

 recnoadr
 recposadr
 simpel_base
 start_restrik
 testa
 testb
 testc
 testparam1
 testparam2
 z_ilængde
 z_itype
 zonestartadr

Procedures: close_soda
 for_which
 getblock
 init_ld_tabeller
 sd_transfer
 soda_block_proc
 sodatestab
 sodatestc
 sodatestproc1
 sodatestproc2
 søg_itypekæde
 søg_itypeliste
 udskriv_d

A. Syntax of the SODA LD language

The following is a complete syntax description of the SODA LD language. While section 3 describes the language by means of examples and verbal explanations, the description here is formal and as exact and precise as possible. The notation used here to express the syntax rules is a slightly modified version of the INFORMAL language described in ref 11. Compared to the reference the concatenation symbol `-*` has been replaced by space.

The principle of the notation is illustrated by the first statements below that expresses the overall structure of an LD description in fig. 3.1. In short, the applied symbols have the following meaning:

```

.=      'is defined as'
!       alternatives
,       elements with optional order
(.)    optional element
(*)    any number of occurrences
(+)    at least one occurrence
' '    a constant syntactical unit
< >   members of a character set
;       termination of statement

```

```

ld_description .= head_line  variable_section
                record_output_input_section(.)
                set_section(.)  endline;

```

The statement declares that an `ld_description` consists of five syntactical units (SU) that must appear in the order as specified. Each non constant SU must be defined in a separate statement.

```
head_line      . = 'local data' ld_ident ':' program_language
                user_spec(.) nl;
```

The SU 'head_line' is composed of six units where the first and third are constant and defined by the characters enclosed by the apostrophes. The SU 'user_spec' is optional as indicated by the suffix (.) In the 'head_line' as well as in other parts of the LD description any number of spaces may appear between syntactical units.

```
ld_ident      . = ld_number '/' ld_name;

ld_number     . = NUMBER (1 to 99);

ld_name       . = identifier;  <* max 17 characters *>
```

The symbol 'NUMBER' with its parameters declares that the SU is an integer in the specified range.

```
program_language . = 'duet' ! 'algol';

user_spec       . = 'user' user_number;

user_number     . = NUMBER (1 to 63);

end_line        . = 'end' ld_number;
```

Apart from 'nl' and 'identifier' which are defined below and 'variable_section' and 'set_section' defined on page A4 and A7 respectively, this ends the formal description concerning fig 3.1. The remaining parts of the formal description will not contain explanatory comments. If necessary, the reader must consult ref 11.

```

<* General definitions *>

nl                . = comment(.) newline_char;

newline_char      . = <10>;

comment           . = '<*' text_char (*);

text_char         . = <ident_char , '+-*/_!?:;,.( )<>=' , 32>;

identifier        . = letter i_group (*);

letter           . = <'abcdefghijklmnopqrstuvwxyæøå'>;

i_group           . = '_' ident_char(+);

ident_char        . = <letter, digit>;

digit            . = <'1234567890'>;

```

NOTE!! Numbers which appear enclosed in the comment brackets <* *> and preceded by the letter A or N, will refer to the page where the SU is defined or to a semantic note on page A12ff, respectively.


```

decimals                . = NUMBER(1 TO 6);

number_of_chars         . = NUMBER(1 TO 767);

number_of_bytes         . = NUMBER(1 TO 511);

result_indicator        . = 'soda'          !
                        'recno'           !
                        IF program_language = 'duet' THEN(
                        'readspec'        !
                        'readterm'        !
                        'error')         ;

arrayspec               . = '(' number_of_elements ')';

number_of_elements      . = NUMBER(1 TO 511);

valuespec_ref           . = 'w' valuespec_no;

normvalue_ref           . = 'n' normvalue_no;

field_equiv             . = '=' field_name;

field_name              . = 'identifier';

alternative_name_spec   . = '(' language_code ')' text_constant nl;

language_code           . = NUMBER(1 TO 9);

value_spectrum          . = 'w' valuespec_no ':' main_type
                        value_segment (',' value_segment) (*) nl;

value_spec_no           . = NUMBER(1 TO 127);

```

```

main_type          .= 'n' ! 'r' ! 't';

value_segment      .= single_value !
                    open_interval !
                    closed_interval;

open_interval      .= ('>' ! '<') single_value;

closed_interval    .= single_value 'to' single_value;

single_value       .= num_constant !
                    short_text !
                    varno !
                    varname ;
                    <*N23*>

num_constant       .= sign(.) principals ('.' decimals)(.);

principals         .= integer;

decimals           .= integer;

short_text         .= '.' short_char short_char(.) short_char(.) '.';

short_char         .= <text_char, ap>;

norm_value_spec    .= 'n' norm_value_no ':' main_type norm_value nl;

norm_value_no      .= NUMBER(1 TO 127);

norm_value         .= IF main_type = 't'
                    THEN (varno ! varname ! text_constant) <*N23*>
                    ELSE single_value;

text_constant      .= ap text_char(*) ap;

```

record_output_input_section

```
.= (record_output_section(.), record_input_section);
```

```
record_output_section    .= descripfile_spec(.)
                          'record output' ':' physical_file nl
                          logical_file_spec(+);
```

```
logical_file_spec       .= logical_file record_spec(.) nl;
```

```
record_input_section    .= descripfile_spec(.)
                          'record input' ':' physical_file nl
                          logical_file_spec(+);
```

set_section

```
.= descripfile_spec(.)
   'record sets' ':' nl set_declaration(*);
```

```
set_declaration        .= set_decl_head(.)
                          set_restriction(.)                <*A8 *>
                          usage_specification(.)            <*A9 *>
                          log_specification(.)               <*A9 *>
                          ident_specification(.)            <*A9 *>
                          mother_specification(.)            <*A10*>
                          daughter_specification(.)          <*A10*>
                          field_specification(.)             <*A11*>
```

```
set_decl_head          .= 's' set_number ':' set_name mother_subscript(.)
                          '=' logical_file file_qualification(.) record_spec(.) nl;
```

```
set_number             .= NUMBER(1 TO 63);
```

```
set_name               .= identifier;
```

mother_subscript . = '(' set_number '); <* N1*>

logical_file . = db_identifier; <* N2*>

file_qualification . = 'in' physical_file;

physical_file . = db_identifier; <* N3*>

record_spec . = '(' record_type_list ');

record_type_list . = record_type (',' record_type)(*);

record_type . = ('i' NUMBER (1 to 9999)) ! <* N4*>
record_type_name

set_restriction . = 'for' 'which'
(general_restriction ! selective_restriction);

general_restriction . = logical_expr nl;

selective_restriction . = (record_type_list ':' logical_expr nl)(+)
('else' ':' logical_expr nl)(.);

logical_expr . = relation !
logical_expr logical_opt logical_expr !
'(' logical_expr ');

relation . = operand rel_opt (operand ! constant) ! <* N5 *>
operand ('-',')(.) 'in' valuespec_ref; <* N6 *>

```

operand          . = field_ref subscript(.) !
                  restr_var_ref subscript(.) ;

field_ref        . = fieldno ! fieldname;

restr_var_ref    . = varno ! 'v.' varname;          <*N23*>

subscript        . = '(' integer ') , ;          <*N8 *>

rel_opt          . = '=' !
                  '<' !
                  '>' !
                  '<>' !
                  '>=' !
                  '<=' ;

constant         . = num_constant !
                  text_constant ;

logical_opt      . = 'and' !
                  'or' ;

valuespec_ref    . = 'w' valuespec_no;          <*N9*>

```

```
usage_specification . = 'usage' ':' usage_element(+) nl;
```

```
usage-element . = 'get' !
                'next' !
                'lookup' !
                'put' !
                'create' !
                'delete' !
                'newset';
```

```
log_specification . = ('log' 'before' ':' access(+) nl) (.)
                    ('log' 'after' ':' access(+) nl) (.);
```

```
access . = 'all' !
           'read' ! 'put' !
           'get' ! 'next' !
           'insert' ! 'update' !
           'newset' ! 'create' !
           'delete';
```

```
ident_specification . = 'ident' 'spec' ':' nl
                      (normal_ident_assoc(+) !
                       recno_ident_assoc);
```

```
normal_ident_assoc . = (varref ! num_constant) '=' ident_field n);
```

```
varref . = simple_var !
          array_var subscript;
```

```

simple_var      .= varno      ! varname;          <*N11,N23*>
array_var      .= varno      ! varname;          <*N12,N23*>
ident_field    .= fieldno    ! fieldname;        <*N13*>
fieldno        .= 'f' NUMBER(1 TO 217-1)      <*N7*>
recno_ident_assoc .= simple_var '=' 'recno' nl;    <*N14*>

```

```

mother_specification .= ('mspec' ' mref_field nl m_spec_line(+)) (+);

```

```

mref_field      .= fieldno    ! fieldname;        <*N15*>

```

```

m_spec_line     .= varref m_ass ident_field nl !    <*N10*>
                 num_constant '->' ident_field nl; <*N16*>

```

```

m_ass           .= <      !
                 ->     !
                 <->    !

```

```

daughter_specification .= 'dspec' ':' nl
                        d_spec_line(+);

```

```

d_spec_line     .= daughter_access_spec !
                 daughter_delete_spec;

```

```

daughter_access_spec .= 's' set_number '<' dref_field nl; <*N17*>

```

```

dref_field      .= fieldno    ! fieldname;        <*N18*>

```

```

daughter_delete_spec .= dref_field;

```



```

field_specification  . = ('fieldspec' fspec_no(.) ':' nl fspec(+)) (*);

fspec_no             . = NUMBER(0 TO 15);

fspec                . = recno_fspec !
                       const_fspec !
                       var_fspec;

recno_fspec          . = simple_var 0< 'recno';                                <*N14*>

const_fspec          . = (num_constant ! short_text)
                       ('>' !
                        '->' !
                        '=>' !
                        '* ' '->' !
                        '* ' '=>' );

var_fspec            . = f_varref !
                       ('-' !
                        '>' !
                        '->' !
                        '=>' !
                        '<' !
                        '<>' !
                        '<->' !
                        '<=>' !
                        '* ' '->' !
                        '* ' '=>' !
                        '* ' '<->' !
                        '* ' '<=>' ) fieldref nl;                                <*N10*>

```

```
f_varref      .= simple_var !  
              array_var subscript(.);  
  
field_ref     .= simple_field      !      <*N21*>  
              array_field field_subscript(.) ! <*N19*>  
              group_no;  
  
simple_field   .= fieldno ! fieldname;  
  
array_field   .= fieldno ! fieldname  
  
field_subscript .= '(' (simple_var ! num_constant) ')'; <*N20*>  
  
group_no      .= 'g' NUMBER(1 TO 9999);      <*N22*>
```

Semantic notes:

1. The set number refers to the mother set which must be declared elsewhere in the LD description.
2. The identifier must be declared as a physical file in the DB description.
3. The identifier must be declared as a logical file (register) in the DB description.
4. The record type must be declared in the DB description as associated with the logical file.
5. The operands/constants associated with a given 'rel_opt' should be compatible in type.
6. The operand must be numeric and compatible with 'main_type' of the referred value specification.
7. The field must be declared in the DB description as a member of at least one of the records of the logical file.
8. The value must not exceed the number of elements in the array.
9. The value specification number must be declared in the value spec. table.
10. The type of field and variable must be compatible.
11. The variable must not be declared as an array.
12. The variable must be declared as an array.

13. The field must be declared as an ident field.
14. The variable type must be 'recno'.
15. The field type must be 'mref'.
16. The type of the ident field must be numeric
17. The referred set must be declared as a mother subscripted set referring the mother set containing the 'daughter_access_spec'.
18. The field type must be 'dref' (list).
19. If a subscript is specified the field must be declared as an array.
20. The variable type must be numeric and the value of 'num_constant' must not exceed the number of elements in the array filed.
21. The field must be a member of at least one of the record types of the current set.
22. The group must be declared in the DB description as a member of at least one of the record types of the current set.
23. Variable number must not be used if variables are declared without explicit variable numbers.

B. Error messages from the SODA LD compiler

An entry in the list below will be composed according to the following frame:

error number: error text
 possible error parameters
 possible further explanation and reaction

1: SYNTAX

The part of the input text under treatment when the error was recognized.

The current line of input text.

A wrong syntactical construction was recognized. The remaining part of the line is skipped. In case of serious errors in a set declaration the remaining part of the set declaration is skipped with a sequence of (possibly) irrelevant syntax errors.

2: SYNTAX

3: SYNTAX

As for 1.

As for 1, but the compiler proceeds with the text of the current line.

4: ILLEGAL ERROR NUMBER

The wrong number

System error: Please contact maintenance group.

5: ILLEGAL END NUMBER

The identifying number of the LD description.

The wrong end number.

The two numbers should be equal.

6: SYNTAX

As for 1.

As for 1.

7: ARRAY INDEX MISSING FOR FIELD/VAR:

The field or variable number.

The arrays may not be used in set restrictions.

8: ILLEGAL DB DESCRIPTION. ERROR TYPE:

Error code:

-1: no DB description file.

-2: illegal version of DB description.

-3: (not used).

-4: other error.

9: UNDECLARED VARIABLE

Variable name and number.

10: VARIABLE PREVIOUSLY DECLARED IN LINE

The line number where the variable was first declared.

11: ILLEGAL VARIABLE NUMBER

The illegal value.

A variable number is outside the legal range.

12: ILLEGAL LANGUAGE CODE

The illegal value.

The language code is outside the legal range (1 TO 9).

13: ILLEGAL USE OF ARRAY VARIABLE

The variable name and number.

An array variable may not be used in the current situation.

14: INCONSISTENT NUMBER OF DECIMALS

The sequence number of the value inside the current line.

The number of decimals in all values should be the same.

15: ILLEGAL NUMBER OF ELEMENTS

The illegal value.

The specified number of array elements is outside the legal range (1 TO 511).

16: ILLEGAL W-NUMBER

The illegal value.

The reference to the value spectra table is outside the legal range (1 TO 127).

17: ILLEGAL N-NUMBER

The illegal value.

The reference to the norm value table is outside the legal range (1 TO 127).

18: ILLEGAL VARIABLE NUMBER

The illegal value.

A variable reference is outside the legal range. (1 TO 2047).

19: ILLEGAL TYPE OF VARIABLE

The variable name and number.

A variable of this type is not allowed in the current situation.

20: W-NUMBER PREVIOUSLY DECLARED IN LINE

The line number where the number was first declared.

21: ILLEGAL W-NUMBER

The illegal value.

The value spectrum number is outside the legal range (1 TO 127).

22: ILLEGAL VALUE TYPE

The sequence number of the value inside the current line.

The value is incompatible with the type of the w-spec/n-spec entry.

23: ILLEGAL LENGTH OF VARIABLE OR TEXT

The illegal value.

The specified length is outside the legal range.

24: ILLEGAL N-NUMBER

The illegal value.

The norm value number is outside the legal range (1 TO 127).

25: N-NUMBER PREVIOUSLY DECLARED IN LINE

The line number where the number was first declared.

26: ILLEGAL SET NUMBER

The illegal value.

A set number is outside the legal range (1 TO 63).

27: SET NUMBER PREVIOUSLY DECLARED IN LINE

The line number where the set was first declared.

28: UNKNOWN LOGICAL FILE

The logical file is not declared in the DB description.

29: DB DESCRIPTION ERROR. LOGICAL FILE

Error in DB description file.

If error is persistent after re-generation of the file, please contact maintenance group.

30: ILLEGAL RECORD TYPE

Record type number.

The record type is not declared for the logical file in the DB description.

31: ILLEGAL RECORD TYPE

Record type number

The record type value is outside the legal range (1 TO 9999) (Temporary restriction in SODA). The reference is ignored.

32: ILLEGAL SECTION FOR LD FILE IN DESCRIPTFILE

Logical file type.

The specified section for the location of the ld file in the physical description file is occupied by a logical file of the type indicated.

33: ILLEGAL NUMBER OF DECIMALS

The number of implied decimals for a variable may not exceed 6.

34: LOGICAL FILE NOT UNIQUE

The logical file name should be qualified by a reference to the proper physical file. The whole set declaration is skipped.

35: UNKNOWN PHYSICAL FILE

The physical file is not declared in the DB description. The whole set declaration is skipped.

36: DB DESCRIPTION ERROR. PHYSICAL FILE

Error in DB description file. If error is persistent after regeneration of the description file, please contact maintenance group.

37: ILLEGAL FILE TYPE

DB physical file type.

The set declaration does not match the type of the associated physical file.

38: CONFLICT BETWEEN LOG AND USAGE SPECIFICATION

39: FIELD SPEC. NUMBER NOT UNIQUE IN SET

The illegal number.

40: USAGE SPECIFICATION MISSING

A statement covering all usage elements is assumed.

41: USAGE OR LOG ELEMENT DOUBLE SPECIFIED

42: COPY NOT IMPLEMENTED

43: EMPTY SPECIFICATION

May appear if all associations in a specification group are erroneous.

44: ILLEGAL INDEX

A variable index or an index at all is illegal in the current situation.

45: ILLEGAL INDEX VALUE

The illegal value.

The declared number of elements in the array.

The index value is outside the range defined by the array.

46: ILLEGAL ASSOCIATION OPERATOR47: ILLEGAL FIELD NUMBER

The illegal value.

SODA cannot accept field numbers outside the range (1 TO $2^{17}-1$).

48: UNKNOWN FIELD

The field number.

The field is not declared for the associated logical file.

49: DB DESCRIPTION ERROR. FIELD

Error code.

See error 29.

50: ASSOCIATION OPERATOR INCONSISTENT WITH USAGE51: SPECIFICATION TYPE ILLEGAL FOR CURRENT SET

Specification type: 1: Ident specification
2: Mother specification
3: Field specification
4: Daughter specification

Set type (internal code: see page 6-10)

52: RECNO ILLEGAL FOR CURRENT SET

The word 'recno' is not allowed for set type M.

53: TOO MANY ASSOCIATIONS IN SPECIFICATION GROUP

For field specifications max 63 is permitted.

For ident specifications with 'recno' only one is permitted.

54: INCONSISTENT USAGE SPECIFICATION55: NOT IDENT FIELD

A field mentioned in the ident specification has not the administrative status ident in the DB description (cf. ref 2).

56: DOUBLE SPECIFIED IDENT FIELD

The same ident field is mentioned twice in the ident specification.

57: ILLEGAL FIELD REFERENCE

The right side of the association must be 'recno'.

58: TYPE CONFLICT

The variable type must be 'recno'.

59: NOT M-REF FIELD

The field in a mother specification line must be declared as m-ref in the DB description.

60: DB DESCRIPTION ERROR. LIST

Error code.

See error 29.

61: NOT D-REF FIELD

The field in a daughter specification must be declared as a d-ref (list) field in the DB description.

62: ILLEGAL LIST NUMBER

SODA cannot accept list numbers outside the range (1 TO 63).

63: IDENT SPEC. MISSING

Set number.

When usage 'get' is indicated or 'create' for set type 1 then the ident specification must not be omitted.

64: ILLEGAL UPDATE OF FIELD

Field status code 1: ident
 2: recno
 3: record length
 4: record type

The association defines the update of a constant field.

65: DB DESCRIPTION FILE CANNOT BE EXTENDED

Error code.

The resources for the description file are insufficient.

66: ILLEGAL DB-DESCRIPFILE REFERENCE

DB file index.

DB file index must not exceed the number of DB descripfiles specified in the activation of the SODALD compiler.

67: DB DESCR. FILE, INSERT ERROR 1

Error code.

System error. Please contact maintenance group.

68: DB DESCR. FILE, CONTROL RECORD ERROR

Error code.

System error. Please contact maintenance group.

69: AUTOMATIC CREATION OF VARIABLE DECL. AREA NOT POSSIBLE

No declaration text is generated.

70: AUTOMATIC CREATION OF LIST AREA NOT POSSIBLE

The translation is performed without listing.

71: LENGTH CONFLICT

Variable length.

Field length.

The receiving location is too short for the value.

72: UNDECLARED DAUGHTER SET

The daughter set number.

73: DAUGHTER/MOTHER SET CONFLICT

The daughter set number.

The daughter specification refers to a set that is not subscripted with the set containing the daughter specification.

74: ILLEGAL DAUGHTER SET

The daughter set number.

The referred daughter set is not mother subscripted.

75: UNDECLARED W-NUMBER

The illegal value.

76: UNDECLARED N-NUMBER

The illegal value.

77: TYPE OF VALUE SPECTRUM NOT COMPATIBLE WITH TYPE OF VARIABLE

Variable name and number.

78: TYPE OF NORM VALUE NOT COMPATIBLE WITH TYPE OF VARIABLE

Variable name and number.

79: INACTIVE ASSOCIATION

The referred field is not declared for any of the record types in the set.

80: ILLEGAL VARIABLE TYPE

The variable type code.

Only simple word variables are allowed for the association operator -

81: TYPE CONFLICT

The left part type code.

The field type code.

The combination of types of the left and the right part is illegal.

82: ARRAY/SIMPLE CONFLICT

Number of elements in the left part.

Number of elements in the field.

The combination of an array with a simple location or an array element is illegal.

83: CONFLICT IN NUMBER OF ELEMENTS

Number of elements in the variable.

Number of elements in the field.

The receiving array has too few elements.

84: ILLEGAL NORM VALUE ASSIGN

The variable type code.

Assign of standard values is illegal for variables of that type.

85: UNDECLARED OR ILLEGAL MOTHER SET

Mother set number.

Only sets associated with CF master files may be mother sets.

86: DAUGHTER SPEC. INCOMPLETE IN MOTHER SET

Mother set number.

The daughter specification in the mother set does not contain an association to the current set.

87: LOGICAL FILE CONFLICT

Mother set number.

Logical file number from daughter spec.

Logical file number for current set.

The logical file derived from the daughter spec. does not match with the logical file of the current set.

88: MOTHER CHAIN CONFLICT

M-ref field number.

The chain associated with an m-ref field in the mother spec. is the same as the chain to the mother set.

89: DB DESCRIPTION ERROR. RECORD

Error code.

See error 29.

90: FIELD REF. MISSING IN DAUGHTER SPEC.

The record type.

The field number.

When usage 'delete' is indicated, a daughter specification must be stated with reference to all d-ref fields in the record type of the set.

91: MOTHER SPEC. MISSING FOR

The record type.

The field number.

When usage 'create' is indicated, mother specifications must be stated for all m-ref fields of the record types in the set except for the one which defines the link to the mother set.

92: ILLEGAL FIELD TYPE

A field of this type is not permitted in the association.

93: ILLEGAL FIELD NUMBER

The illegal value.

SODA cannot accept field numbers outside the range (1 TO $2^{17}-1$).

94: UNKNOWN FIELD

The logical file number.

The field number.

The field is not declared in the DB description for the associated logical file.

95: DB DESCRIPTION ERROR. FIELD

Error code.

See error 29.

96: ILLEGAL FIELD TYPE

The field number.

A field of this type is not allowed in restrictions.

97: ILLEGAL VARIABLE NUMBER

The illegal value.

The variable number is outside the legal range.

98: UNDECLARED VARIABLE

The variable number.

99: ILLEGAL VARIABLE TYPE

The variable name and number.

100: ILLEGAL RECORD TYPE

Record type number.

The record type is not declared for the logical file in the DB description.

101: RECORD TYPE USED PREVIOUSLY IN RESTRICTION

The record type number.

102: ILLEGAL W-NUMBER

The illegal number.

The number is outside the legal range. (1 TO 127)

103: UNDECLARED W-NUMBER

The w-number.

104: ILLEGAL TYPE OF VALUE SPECTRUM

The w-number.

A value spectrum of type text cannot be used in set restrictions.

105: ILLEGAL INDEX VALUE

Index value.

The number of elements in the array.

The index value is outside the range defined by the array.

106: FIELD UNDECLARED IN RECORD TYPE

The field number.

The record type number.

107: TYPE CONFLICT

The combination of types for the two operands of a relation is illegal.

108: NUMBER OF DECIMALS IN VALUE SPECTRUM NOT MATCHING VARIABLE

The variable name and number.

109: NUMBER OF DECIMALS IN NORM SPECTRUM NOT MATCHING VARIABLE

The variable name and number.

110: LENGTH OF NORM VALUE TEXT TOO LONG FOR VARIABLE

The variable name and number.

111: NORM VALUE EXCEEDING RANGE OF VARIABLE

The variable name and number.

112: ILLEGAL USER NUMBER

The illegal value.

The user number is outside the legal range (0 TO 127).

113. LD IDENTIFIER TOO LONG.

The identifier is truncated to 17 characters.

114: DOUBLE DECLARED VARIABLE IDENTIFIER

The variable name has been used in a previous declaration.

115: AGGR. AND ELEMENTARY FIELD IN SAME IDENT SPEC.

The field number (of elementary field).

The ident specification may not contain associations both for an aggr. field and an element of the aggregate.

116: IDENT FIELD MISSING

The field number.

The ident specification must contain associations for all ident fields.

117: LOGICAL AND PHYSICAL FILES NOT ASSOCIATED

The whole set declaration is skipped with (possibly) irrelevant error messages in each line.

118: ILLEGAL FIELD SPEC. NUMBER

The illegal number.

The number is outside the legal range (0 TO 15).

119: ILLEGAL VARIABLE REFERENCE

The illegal variable name and number.

A variable is referred from an entry in the value spectra or norm value table which is referred from the variable itself.

120: VARIABLE NUMBER NOT ALLOWED

The variable number.

A variable number must neither be referred to nor declared, if the first variable was declared without variable number.

121: NOT ALLOWED WHEN VARIABLES ARE NUMBERED

A set must not be declared by 'record output' or 'record input', if variables are declared with variable numbers.

122: SYNTAX

As for 1.

A syntactical error in the set declaration line. The remaining part of the set declaration is skipped without further check.

123: TOO MANY VARIABLES

The variable number of the last read variable.

The compilation is finished immediately.

124: ILLEGAL CHARACTER

The last read syntactical unit contains an illegal character.

125: UNKNOWN SYNTACTICAL UNIT

The last read syntactical unit cannot be recognized.

126: TOO MANY DIGITS IN NUMBER

The number.

The number of digits in a numerical constant must not exceed 15, incl. decimals. And the greatest possible number is 140 737 488 355 327.

127: TOO MANY DECIMALS IN NUMBER

The number.

The number of decimals in a numerical constant must not exceed 6.

128: Not used

129: Not used

130: FIELD NAME WITHOUT PREFIX

The logical file number.

The field number.

All fields used for record output or record input must contain a prefix in the field name and the first character after the prefix must be a letter (cf. section 3.3.2).

131: FIELD TYPE ILLEGAL

The logical file number.

The field number.

The field type.

132: FIELD/VARIABLE TYPE CONFLICT

The logical file number.

The field number.

A variable with the same name has previously been declared with another type.

133: REPRESENTATION CONFLICT

The logical file number.

The field number.

A variable with the same name has previously been declared with another number of decimals or characters.

134: DIMENSION CONFLICT

The logical file number.

The field number.

A variable with the same name has previously been declared with another array specification (dimension).

135: NORM VALUE CONFLICT

The logical file number.

The field number.

A variable with the same name has previously been declared with another norm value (or possibly with no norm value).

136: ILLEGAL NUMBER OF DECIMALS

The logical file number.

The field number.

The number of decimals in SODA must not exceed 6.

137: ILLEGAL NUMBER OF ELEMENTS

The logical file number.

The field number.

The number of elements of any array in SODA must not exceed 511.

138: RECORD TOO LONG

The record type number.

The record length in SODA must not exceed 1024 halfwords (2 segments).

139: ILLEGAL REPRESENTATION

The logical file number

The field number.

The number of characters in a text variable in SODA must not exceed 255.

140: REDEFINED RECORD TYPE

The record type name.

The record type has been specified previously.

141: ILLEGAL RECORD TYPE

The record type name.

The specified record type does not belong to the logical file.

142: RECORD INPUT NOT ALLOWED FOR DATA ENTRY143: UNDECLARED VARIABLE

The variable name.

144: UNDECLARED VARIABLE

The variable name.

145: VARIABLE NUMBER NOT ALLOWED

The variable number.

See error 120.

146: ILLEGAL FIELD TYPE

The field type code.

The field number.

A field of this type must not be used in the declaration of an associated variable.

147: ILLEGAL FILE TYPE

The file type code.

The physical file in a record output or a record input set must be of type 'outvar'.

148: NO PERMANENT VARIABLES SPECIFIED

Only used for Data Entry.

149: DB DESCRIPTION ERROR. RECORD TYPE

Error code.

See error 29.

150: UNKNOWN VREFNUMBER

The vref-number.

Inconsistency in the DB description. The value reference number of a field is undeclared.

151: DB-DESCRIPTION ERROR. VALUE SECTION

Error code.

See error 29.

152: Not used.

153: ILLEGAL USAGE SPECIFICATION

Only used for Data Entry.

C. References

1. Connected Files System RCSL 28-D5
2. DATABASE80 RCSL 21-V031
3. P. Lindgreen, E. Rosenberg: SODA - A Flexible Scheme for Database/Program Interface.
Proceedings of the International Computing Symposium 1977
North Holland Publ. Co
4. BS-System RCSL 31-D288
5. DUET RCSL 21-V046
6. DES80 - SODA LD RCSL 21-V018
7. CODASYL DBTG April 71 Report
8. ALGOL 6 RCSL 31-D322
9. BOSS (users manual) RCSL 31-D108
File Processor RCSL 55-D21
10. SYSDOK RCSL 28-V033
11. P. Lindgreen: INFORMAL - A Comprehensive Method for Input Data Description. Proc. Norddata, Helsinki 1976
Also see: INFORMAL RCSL 28-D17

Alphabetical index

An (a) specified for a term indicates a reserved ALGOL identifier in the SODA DBMS.

An (fp) specified for a term indicates an FP-parameter keyword in activation of the LD compiler.

adate field	81
Administrative status (of field)	81
Aggr. field	50, 55f, 72, 82
ALGOL	3, 9, 10, 32, 35f, 38, 90, 117, 133ff, 139, 144, 148
Algol declarations	10, 11, 38, 90, 100, 111
Alternative var.name	38f
Anonymous var.number	40
Application program	3, 8, 13, 15, <u>133ff</u>
Array field	see Field array
Array variable	see Variable array
Associated variable decl.	<u>37</u> , 81, 107
Association line	see Field/variable ass.
Association symbol	54f, 60ff, 63f, 68ff, 79, 86, 108
Asterisk (*)	38, 70, 84
beskrivnavn (a)	137
Bits variable	<u>36</u> , 50, 55f, 72
BS file	13, 56, 73, 77, 85
CF list file	13, 21, 48, 56, 73
CF master file	13, 55
Character constant	see Short text
Checksum field	81
CLOSE_FILE	115, <u>132a</u>
close_soda (a)	134, 136, 145
CODASYL	1
Command unit	see FP-parameters

Comment	31
Concurrent access	4
Connect	58f
Copyrecord	81, 86
CREATE	14, 18, 29f, 37, 43, 54, 58ff, 66, 69, 70, 73, 115, 124, <u>127f</u> , 142, 145
Cross reference list	97, 98, 107f
Current record	<u>19f</u> , 36, 58, 60, 119, 121, 124
Database80	8 (see also DB description)
Data independence	2
Data protection	4
Date variable	35, 72
Daughter record	21ff, 24, 57f, 60, 64, 129f
Daughter set	24, 48, 58
Daughter specification	46, 48, 63ff
DB currec	19, 124, 129, 146
DB description	1, 4, 11, 13, 18, 20, 21, 24, 37, 45 46, 55, <u>87</u>
DBMS	see SODA DBMS
Declaration file	see Algol declarations
DELETE	15, 19, 64, 115, <u>129f</u> , 142, 145
descripfile (fp)	87, 96, 107, 110
Direct access	15, 48, <u>54</u> , 56
d-ref field	21, 50, 63ff, 82
dspec	see Daughter specification
DUET	3, 9, 10, 32, 35f, 38, 40, 90, 116f, 133, 139, 144
DUET errors (DBMS)	118ff, 144
End line	32
Equivalence field	83
error (result.var)	36
Error messages	91, 106, 112ff, app.B
Error procedure (DBMS)	117, <u>144ff</u>
Explicit variable decl.	<u>35</u> , 81, 107

Field access	2, 27
Field array	56, 71, 80
Field group	82
Field name	37, 55, 80
Field number	55
Field reference	37, 55, 62, 64, 67, 70
Field specification	29, 46, 57, <u>66ff</u> , 119, 122, 124f, 127
Field/variable association	27ff, 55, 57, 60ff, 63, 66ff, 77, 86
filnr (a)	<u>143</u> , 145
for_which	see Set restriction
FP parameters	92ff
fspec	see Field specification
GET	14ff, 22, 30, 54, 69, 70, 115, <u>119f</u> , 124, 142, 145
Group	see Field group
Head line	32
Ident fields	14f, 30, 55, 56, 81
Ident specification	30, 46, <u>54ff</u> , 119, 123, 127, 131
Implicit variable decl.	79, 80, 107
init (fp)	95
init_soda_1 (a)	134ff, <u>137</u> , 144
init_soda_2 (a)	134ff, 137, <u>140</u>
i-number	see Record type
Insert location	18
Language code	39, 100
LD compiler	see SODA LD compiler
LD description	1, 2, 8f, 11f, 13, 15, 18, 29, <u>31</u>
LD file	10ff, 32, 88, 90, 96, 110, 136f, 140, 143

LD identification	see LD number
LD number	32, 104, 151
ld_afsnit_nummer (a)	137
ld_brugernr (a)	138
ld_ident (a)	138
ld_initialer (a)	139
ld_navn (a)	139
ld_regdato (a)	139
ld_regtid (a)	139
ld_varsum (a)	139
ld_version (a)	138
ldfile (fp)	96
ldtext (fp)	94
Line numbers	105, 113
List field	see d-ref field
List file	see CF list file
list (fp)	97
Listing (of LD descr.)	11, 70, 83, 90, <u>104</u> , 114
listout (fp)	97, 98
Logical expression	49f
Logical file	1, 20f, 46f, 78
Log procedure	53, 141ff (see also soda_log)
Log specification	46, <u>53</u>
LOOKUP	14, 115, <u>123</u> , 142, 146
Master file	see CF master file
Mother/daughter relation	63, 116
Mother record	21ff, 24, 58, 63f, 119, 122, 125f
Mother set	22, 26, 46ff, 57ff, 63
Mother specification	46, 48, <u>57ff</u> , 119, 122, 125, 127
Mother subscription	22ff, 45ff, 48, 58, 59, 63ff
m-ref field	50, 57ff, 82
mspec	see Mother specification
names (fp)	100
New currec	19, 124f, 129, 146

New page	104
NEWSET	17, 24, 48, 54, 63, 115, 116, 121, 124, 128, <u>131</u> , 142, 145, 146
NEXT	14, 16f, 36, 69, 70, 86, 115, 121f, 124, 142, 146
Norm value reference	<u>37</u> , 43, 80, 107
Norm value table	33, 37, <u>43ff</u> , 70, 127
n-ref	see Norm value reference
Numeric constant	42, <u>44</u> , 55, 70, 71
Numeric variable	35, 55
ok bit	114
Operational var.name	38f, 100
paper (fp)	97, 99
Parameter group	see FP parameters
Physical file	2, 13, 20, 53
Prefix (in field name)	80, 82
prim (a)	136
Procedure number	see sd_procuo
PUT	15, 18, 22, 29f, 36, 58ff, 69, 71, 81, 115, <u>124ff</u> , 127, 142, 145
readspec (resultvar)	36
readterm (resultvar)	36
reclength field	81, 86
recno	<u>16</u> , 22, 30, <u>36</u> , 50, 55, 56, 67, 73, 116f
recno (resultvar)	36f
Record access	1
Record field	8, 37
Record input	77, <u>85</u>
Record membership	<u>20</u> , 47, 49ff, 119, 121, 123
Record number	see recno
Record output	<u>77</u> , 80
Record set	1, 8, 9, <u>13</u> , 20, <u>45</u>
Record status	19, 116, 124, 129, 131

Record type	20, 46f, 78, 83, 127
rectype field	81, 86
Relation	49f
Repeating group	29, 71, <u>72f</u> , 82
Repeating group vector	72
Repeating group no.of. repetitions	73
Reserved identifiers	137ff, 148ff
Result variables	36, 116f
Rpg	see Repeating group
Scan	see Sequential access
Schema	1
sd_alarm (a)	144
sd_extend_buf (a)	138
sd_fejl (a)	146
sd_kædeindex (a)	143
sd_log_before (a)	143
sd_procno (a)	<u>142</u> , 145
sd_resultat (a)	146
sd_sprogkode (a)	139
sd_sætnr (a)	<u>142</u> , 145
sd_sættype (a)	<u>143</u> , 145
sd_tilstand (a)	146
sd_transfer (a)	147
sdtrans (a)	136
sdz (a)	143
Secondary mother	59
section (fp)	94, 97
Selection expression	see Set restriction
Sequential access	16f, 116
Sequential file	see BS file and CF list file
Set declaration	15, 17, <u>20</u> , 22, 26, <u>45ff</u>
Set decl. head	46ff
Set number	47, 78, 85
Set reference	48, 63f
Set restriction	20, 41, 46, <u>49</u>
Set sequential position	<u>17</u> , 18, 121, 131

Set sequential status		<u>17</u> , 121, 124, 127, 129, 131f, 146
Set type		<u>14</u> , 16, 117, 143
Short text		42, 71
Singular set		22
size (fp)		1o1, 111
SODA DBMS		1, 1off, 14ff, 115ff, 133, 135, 137
SODA LD compiler		9, 11, 54f, 89ff, 115, 136
SODA LD language		9
SODA LD ...		see also LD ...
SODA program texts		134
SODA variable		8, 27f, <u>33ff</u>
soda (resultvar)		36
soda_bsaccess		139
soda_cfaccess		139
soda_error (a)		117, 134, 136, <u>144f</u>
soda_giveup (a)		139
soda_log (a)		134, 136, 141ff (see also Log procedure)
sodatext 1 (a)		134ff, 144
sodatext 2 (a)		134ff
sodatext 3 (a)		134
SQ file		see BS file
Standard value		see Norm value table
Sub schema		1
sysdok (fp)		<u>94</u> , 1o5, 11o
tduetcode (a)		136
test (fp)		1o2
testout (fp)		1o2
Text variable		35, 72
Transfer (field/var.)		see Field/variable ass.
Unnumbered variables		4o, 77
Usage specification		46, <u>52ff</u> , 62, 65, 68, 71, 115
Value spec reference		<u>37</u> , 41, 49, 8o, 1o7

Value spectra table	33, 37, <u>41ff</u> , 49
vardecl (a)	100, 134, 136 (see also Algol declarations)
vardecl (fp)	100
Variable	see SODA variable
Variable array	37, 83
Variable/field association	see Field/variable ass.
Variable name	34, 40, 51, 55
Variable number	<u>34</u> , 40, 55
Variable reference	40, 42, 43f, 49ff, 55, 61, 67, 70, 107f
Variable table	33ff
Variable type	35, 80
version (fp)	94
warning bit	114
W-ref	see Value spec reference
xref (fp)	97, 98, 107f
zone record	27