Kertész Sándor
Hungarian Meteorological Service
kertesz.s@met.hu
2001-11-8

A short overview of ODB

1. Basic structures

ODB (Observational DataBase) is a newly developed database system to store and handle all kind of observations. It allows fast data extraction by data query language **ODB/SQL**. ODB consists of uniquely named data structures called **tables**. Each table is made up of several uniquely named data entries called **columns**. For example, table HDR contains the report header information (observation type, station ID, geographical coordinates etc.), table BODY contains the information about one observed data (variable type, observed value, observation height etc.) and table ERRSTAT contains observation error for the observed values. The notation for a given column is: *column_name@table_name*. ODB has several data types for columns:

- int, real4, real8: unpacked types
- pk5int, pk9real etc.: packed types, packing for 2, 5 or 9 bits both for integer and real
- string: 8 bytes of character data
- YYYYMMDD and HHMMSS: types to store date
- @LINK: type to ensure relation between tables
- OBSCHAR_t, STATUS_t etc.: compound data types that are used for bitfileds (with maximum of 32 bit members). The notation for a given member in a bitfield type column is: *column_name.member_name*.
- Arrays can be composed of any data type and an array with n element can be defined as *column_name*[n]. The notation for bitfield members of arrays of compound types is: *column_name*[n]→*member_name*.

A set of tables forms a **complete data definition** (these will be called in this document **database types**). At present several types exist. Regarding the data assimilation the most important ones are as follows:

- **ECMA**: the name stands for extended CMA, and it consists of all the tables needed by screening (see Figure 1.2. for the structure of ECMA)
- **ECMASCR**: it is basically another name to an ECMA-type ODB. It could be used for special ODB related programs.
- **CCMA**: the name stands for compressed CMA, and it consists of all the tables needed by configuration e131. Usually, CCMA is created from ECMA after the screening using only the active data (see Figure 1.2. for the structure of CCMA)

Data arrangement in ODB can be imagined as arrays of tables of the same type. The relation between tables is ensured by **@LINK** type columns. For example table HDR has a @LINK type column called body and it contains the information to identify those BODY tables which are connected to the given HDR table, i.e. it shows where the observations of a report are in ODB (Figure 1.1.). Table HDR, of course, needn't be the highest layer in a given ODB type. For example in an ECMA-type ODB on the top there is table DESC, which specifies the analysis date and it has a link to table INDEX that contains information about the timeslots (for 4DVAR) and about the sets. Table INDEX has a link to table HDR and so on. The complete table hierarchy can be seen on Figure 1.2.).

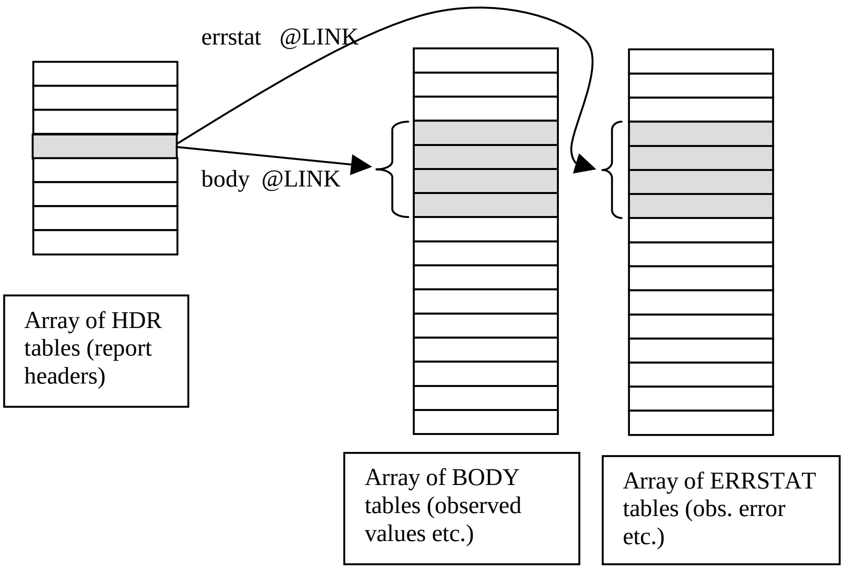

Figure 1.1.
A schematic picture of relations between some tables in ECMA

```
//  CCMA table hierarchy:
//   1  desc
//   2  +--->ddrs
//   3  +--->index
//   4  |     +--->hdr
//   5  |     |     +--->sat
//   6  |     |     |     +--->atovs
//   7  |     |     |     |     +--->atovs_pred
//   8  |     |     |     +--->tovs
//   9  |     |     |     |     +--->rrtovs_pred
//  10 v      v     v     v     +--->rtovs_slev
//  11 |      |     |     |     +--->rtovs_mlev
//  12 |      |     +--->body
//  13 |      |     +--->errstat
//  14 |      |     +--->update
```

```
//  @LINKs with maximum jump of
//   one ("one-loopers"):
//  -1index=hdr
//  -1hdr=sat
//  -1sat=(atovs,rtovs,rtovs,ssmi,scatt,satob)
//
//  Aligned tables:
//  -Abody=(update,errstat,rtovs_body,scatt_body,
//               ssmi_body)
//  -Artovs=(rtovs_slev,rtovs_pred)
//  -Aatovs=atovs_pred
//  -Assmi=ssmi_slev
```

```
//  ECMA/ECMASCR table hierarchy:
//   1  desc
//   2  +--->ddrs
//   3  +--->index
//   4  |     +--->hdr
//   5  |     |     +--->sat
//   6  |     |     |     +--->satob
//   7  |     |     |     +--->scatt
//   8  |     |     |     |     +--->scatt_body
//   9  |     |     |     +--->ssmi
//  10 v      v     v     v     +--->ssmi_slev
//  11 |      |     |     |     +--->ssmi_mlev
//  12 |      |     |     |     +--->ssmi_body
//  13 |      |     |     +--->atovs
//  14 |      |     |     |     +--->atovs_pred
//  15 |      |     |     +--->rtovs
//  16 |      |     |     |     +--->rtovs_pred
//  17 |      |     |     |     +--->rtovs_slev
//  18 |      |     |     |     +--->rtovs_mlev
//  19 |      |     |     |     +--->rtovs_body
//  20 v      v     +--->body
//  21 |      |     +--->errstat
//  22 |      |     +--->update
```

Figure 1.2.
Table hierarchy in ODB for CCMA- and ECMA-type databases

The different tables can be enforced to be **aligned** or **one-looper**. The number of elements for arrays of aligned tables must be the same, and there must be a one to one correspondent between their elements. For example in the ECMA-type database BODY and UPDATE are aligned tables, thus the n-th UPDATE table and the n-th BODY table refer to the same observation (as it is suggested by Figure 1.1.). If tables are enforced to be one-loopers then a master table acts like a middle-man between slave tables. In this case the link (from the master table to a given slave table) length must be 0 or 1. One example for one-looper tables are SAT (master) and ATOVS, RTOVS, RTOVS, SSMI, SCATT, SATOB (slaves).

Data partitioning for multi-processor runs can be realised inside ODB via the so-called **pools**. Each pool contains only a certain part of the database. Partitioning can be done with respect to latitude-bands, or time-slots or other ways. For multi-processor runs the number of pools must be greater or equal than the number of PEs.

A.2. Views

A subset of ODB data can be extracted by data query language ODB/SQL. This is done via the so-called **views**. A view can be defined in the way as follows:

```
SET $airep=2;
CREATE VIEW visu AS
SELECT
   obstype, lat, lon
   varno, obsvalue
   fg_depar
FROM hdr, body, update
WHERE (lat > 0)
    AND obstype=$airep;
```

Variable definition and init. for parametrized query

Desired columns

Name of the view

Special query options

List of tables to be searched

The result of the data retrieval by this view can be imagined as a matrix. This matrix has 6 columns corresponding to the 6 columns given in the SELECT part of the view, and it has as many rows as the number of the observed values in AIREP reports on the northern hemisphere. For each database type there are several predefined views, these are located in the ODB system directories (see next chapter). Views should be placed into **.sql** files. On compilation SQL files are first transformed into C-language code.

A.3. Database definition

The latest version of the ODB system (that I have worked with) can be found on kami under the **~marp001/pub/cy24t1_op2.01/src/main/odb** directory. In what follows this directory will be referred as ODB_sys. In ODB_sys directory **ddl** contains the most important definitions, namely:

- all the different views (in **.sql** files)
- file **CMA.inc** containing the definition of views and tables
- files mdi.inc, obstype.inc etc. containing the definition of the compound data types

For each database type there is an individual directory in ODB_sys with the name of **ddl.***db_type* (ddl.ECMA etc.) and there is a file called *db_type***.dep** in each of them that contains all the tables and views of which the given database is made up. These **ddl.***db_type* directories also contain all the necessary files from directory ddl to build the database (there are links to files in directory ddl).

4. How ODB looks like?

An ODB database physically is nothing else than a directory with two ASCII files and one or more subdirectories. This directory can have an arbitrary name and it can be anywhere in the filesystem. The two ASCII files are as follows:

- *db_type***.dd** (ECMA.dd etc.), that contains the default values for all the columns and variables (for parametrized queries) in the database
- *db_type***.sch** (ECMA.sch etc.), that contains the scheme to create the database

The subdirectories represent the different pools. The name of such a directory is the number of the given pool (pool numbering starts at 1). A pool contains one file for each table of the given database. Basically these files store the data of the database.

As it was mentioned in Chapter 1 ECMA-type and ECMASCR-type databases are basically the same, only their names are different. Thus, for example to create an ECMA-type database from an ECMASCR-type database only ECMASCR.dd and ECMASCR.sch files have to be renamed to ECMA.dd and ECMA.sch.

5. Where to find an ODB for a given date?

At Meteo France observations in ODB format are available since September 1, 2001. For date *yyyymmddhh* (*hh* without leading zero) a global ECMASCR-type ODB with one pool can be found on **delage** as:

/chaine/mxpt/mxpt001/arpege/oper/assim/*yyyy***/***mm***/***dd***/r***hh/***ecmascr.tar**

This tar-file contains the .dd and .sch files and the pool (its name is 1). For former dates OBSOUL or OBSOB observation files should be converted into CMA file and then into ODB. One example-script for this is ~mrpm624/devAL15/example/obs2lamodb.run on kami.

6. ODB in ARPEGE/IFS

ARPEGE/IFS can reach ODB through an intermediate layer of subroutines. The highest-level subroutines are **OPENDB**, **CLOSEDB**, **GETDB** and **PUTDB** (their sources can be found in ODB_sys/cma2odb). The usage of ODB in the model starts with OPENDB, which opens and initialise the database. Then subroutines working with observations have to call GETDB, which extracts data from ODB into temporally allocated matrices. Subroutines then work on these matrices and in the end they call PUTDB, which deallocates the matrices and puts back the data into ODB. Thus, observations are available only between these GETDB-PUTDB calls. Several GETDB-PUTDB calls can follow each other while in the end CLOSEDB closes the database and updates the modified data (on demand).

After this short introduction a detailed description of the whole process will be given. As it was mentioned the usage of ODB starts with **OPENDB** which is used to connect to the database (it is called in subroutine READOBA).

Arg.	**Type**	**Description**
1.	character	Type of ODB (‘ECMA’, ‘CCMA’ etc.)
2.	character	Status of ODB: • for new databases ‘NEW’ • for existing, but updateable databases ‘OLD’ • for existing, but read-only databases ‘READONLY’
3.	integer(32)	Information array. Its first element gives the number of pools.
4.	integer	Number of active elements in array in argument 3. It can be even zero.

Table 6.1.
Arguments of subroutine OPENDB

After calling OPENDB the database is accessible, but no data has been read yet. To extract data from ODB subroutine **GETDB** must be called. Subroutine GETDB reads data – the way of reading is specified in the arguments, mainly through the context - into matrices defined in module **YOMDB**. These matrices can be as follows:

- **MOBHDR**: report header data (integer4)
- **MOBODY**: report body data (integer4)
- **MOBSU**: setup related data (integer4)
- **ROBHDR**: report header data (real8)
- **ROBODY**: report body data (real8)
- **ROBSU**: setup related data (real8)
- **ROBDDR**: DDR related data (real8)
- **SATHDR**: satellite specific header data (real8)
- **SATBODY**: satellite specific body data (real8)
- **SATPRED, SATSLEV, SATMLEV**: misc. satellite specific data

The matrices are allocated on reading and both of their dimensions are set dynamically. The data extraction is not based on single views but on special group of views called **contexts** (see next chapter for details on contexts). A context for each of its views additionally determines the target matrix (one in the list above) and specifies those columns in the view, which should be updated. The first dimension of the matrices refers to the rows in the retrieval, while the second dimension refers to the different columns defined by the given view. Each column in ODB - except for the links - has an integer variable counterpart defined in YOMDB. The GETDB call sets these variables and they can be used as column indices for the matrices. For example, the corresponding value for lat@HDR in YOMDB is MDBLAT and if a view containing lat@HDR is read into ROBHDR, then ROBHDR(·,MDBLAT) will refer to the retrieved lat@HDR values. The exact relation between database columns and YOMDB variables is defined in ODB_sys/cma2odb/initmdb.F90.

Arg.	Type	Description
1.	character	Name of the context
2.	integer	Version of the context
3.	integer	Return value. The number of rows in the extracted data.
4.	integer arr.	Information array.
5.	integer	Number of active elements in array in argument 4.
6.	real arr.	Information array. Not used now.
7.	integer	Number of active elements of array in argument 6 (=0 now)
8.	integer	Number of the desired set or 0 if all the sets are needed.
9.	integer	Number of the desired timeslot or –1 if the value defined in the view should be used (if its not set in the view then all the timeslots will be read).
10.	integer	Number of the desired pool or –1 if the value defined in the view should be used (if it is not set in the view then all the pools will be read).
11.	integer	The desired observation type (obstype@HDR) or –1 if the value defined in the view should be used (if it is not set in the view then all the possible observation types will be read).
12.	integer	The desired observation code type (obschar.codetype@HDR) or –1 if the value defined in the view should be used (if it is not set in the view then all the possible codetypes will be read).

Table 6.2.
Arguments of subroutine GETDB

Another important variable in YOMDB is array **MLNKH2B**. This array - if it is ensured in the given context - corresponds to the link between table HDR and table BODY. Supposing that report headers are read to MOBHDR and report bodies to MOBODY, then MOBHDR(n,·) will contain the header of the n-th report, while the observed values (etc.) connected to this report will be in MOBODY(i,·) where i=MLNKH2B(n),…,MLNKH2B(n+1)-1. The code fragment below from subroutine VERCO illustrates this.

```
!  2.      LOOP OVER THE OBSERVATIONS
OBS_LOOP: DO JOBS = 1, MLNSET(KSET)
    !  3.0   FIRST LOOP OVER THE REPORT
    FIRST_LOOP: DO JNLV = MLNKH2B(JOBS),MLNKH2B(JOBS+1)-1
       !3.1   VARIABLE AND THE MAXIMUM FLAGS
       ICMVNM = MOBODY(ILEV,MDBVNM)
       …
   ENDDO
ENDDO
```

Figure 6.1.
An example from subroutine VERCO

After finishing the work on the observations subroutine **PUTDB** should be called. This subroutine deallocates the matrices and puts back the modified data into ODB. Its arguments are as follows:

Arg.	Type	Description
1.	character	Name of the context
2.	integer	Version of the context
3.	integer	Return value.
4.	integer arr.	Information array. Not used now.
5.	integer	Number of active elements in array in argument 4 (=0 now)
6.	real arr.	Information array. Not used now.
7.	integer	Number of active elements in array in argument 6 (=0 now)

Table 6.3.
Arguments of subroutine PUTDB

Finally, when all the work with observations has already done, **CLOSEDB** deconnects from the database. It is very important to set its arguments carefully. Its arguments are as follows:

Arg.	Type	Description
1.	logical	If it is .TRUE. then those columns that are marked for update in the context will be saved from memory into the database files. Otherwise no modification will be put into ODB!!

Table 6.4.
Arguments of subroutine CLOSEDB

7. Contexts

Contexts contain most of the information needed for GETDB to perform the data extraction. A context basically is a group of views. For each of its views a context determines the target matrix in YOMDB (the result of the data query will be put here) and specifies the columns which can be updated. The definition of contexts can be found in **ODB_sys/cma2odb/ctxinitdb.F90**. In this chapter a basic introduction will be given for contexts via the definition of context SUFGER. The code fragment in Figure 7.1. has been taken from ctxinitdb.F90.

It is obvious from the example that context SUFGER can have several versions and each version consists of four views. The target matrices (in YOMDB) of the views are specified via the **…ncase** variable (JPCASE_ROBHDR means ROBHDR etc.). However, there is something tricky in this definition. View sufger_mobhdr_… should be read into MOBHDR, but **…view(2)%execute=.FALSE.** means that it needn't extract data for this view from ODB, because MOBHDR will contain the same data as the matrix of the view specified in **…view(2)%shared_index**. So in this case MOBHDR will be the integer equivalent of the ROBHDR. This solution reduces the data query time. It is important to know that view sufger_mobhdr_… even needn't exist, and it does not exist indeed (there is no sql file for sufger_mobhdr_… in ODB_sys/ddl). The situation is similar for view sufger_mobod_...

Assignment **view(2)%mlnkh2b=4** ensures the creation of array MLNKH2B that will describe the relation between the matrix of the 2nd and the 4th view, i.e. between MOBHDR and MOBODY.

The last part of the context definition specifies the columns to be updated, i.e. their modified values will be saved into the database.

Loop for the different versions of the context

Name of the view

Specifies the matrix of the view in YOMDB

Ensures the creation of MLNKH2B array between MOBHDR and MOBODY

Columns to be updated, i.e. their values will be saved back into the database

```
else if (cdretr == 'SUFGER') then
if (LSCREEN .and. LLECMA) then
  do j=1,size(sufger_nviews)
   CALL next_context('SUFGER', version=j, nviews=sufger_nviews(j))

   write(ctx(idctx)%view(1)%name,'(a,i1)') 'sufger_robhdr_',j
   ctx(idctx)%view(1)%ncase = JPCASE_ROBHDR

   write(ctx(idctx)%view(2)%name,'(a,i1)') 'sufger_mobhdr_',j
   ctx(idctx)%view(2)%ncase = JPCASE_MOBHDR
   ctx(idctx)%view(2)%execute = .FALSE.
   ctx(idctx)%view(2)%shared_index = 1
   ctx(idctx)%view(2)%mlnkh2b = 4

   write(ctx(idctx)%view(3)%name,'(a,i1)') 'sufger_robody_',j
   ctx(idctx)%view(3)%ncase = JPCASE_ROBODY

   write(ctx(idctx)%view(4)%name,'(a,i1)') 'sufger_mobody_',j
   ctx(idctx)%view(4)%ncase = JPCASE_MOBODY
   ctx(idctx)%view(4)%execute = .FALSE.
   ctx(idctx)%view(4)%shared_index = 3

   if (j == 1) then
     isize = 2
     allocate(ctx(idctx)%view(3)%update_col(isize))
     ctx(idctx)%view(3)%update_col(1) = 'fg_depar@update'
     ctx(idctx)%view(3)%update_col(2) = 'fg_error@errstat'
   else ! No updates otherwise
      ctx(idctx)%view(1:sufger_nviews(j))%cancel = .TRUE.
      ctx(idctx)%view(1:sufger_nviews(j))%putdb = .FALSE.
   endif

  enddo
 endif
```

Figure 7.1.
Definition of context SUFGER

8. How to write a simple ODB reading program?

In this chapter the creation of a simple ODB reading program will be explained in details. All the necessary script and programs for the following example can be found on **kami** in directory **~mrpm624/odb/myodbr**. This directory contains a compiler script (**myodbr.comp**) and an execution script (**myodbr.run**). Directory **src** contains all the sources that are needed:

- **myview.sql**: definition of the view
- **ctxinitdb.F90**: definition of the context. It is a simplified and modified version of the subroutine mentioned in Chapter 7.
- **myodbr.F90**: main program

To define the way of data extraction, first of all a view is needed. This will be a very simple one and it will only retrieve the geographical coordinates for all the AIREP reports:

```
CREATE VIEW myview AS
SELECT
  lat, lon
FROM hdr
WHERE (obstype == 2)
```

Supposing we want to read the result of the query for view **myview** into matrix ROBHDR via context MYCONTEXT, then the following lines should be added in ctxinitdb.F90 to the context definitions:

```
else if (cdretr == ‘MYCONTEXT') then
  CALL next_context('MYCONTEXT', version=0)
  ctx(idctx)%view(1)%name = 'myview'
  ctx(idctx)%view(1)%ncase = JPCASE_ROBHDR
```

The main program firstly opens the database, then extracts data from database and writes them all into the display and finally closes the database without any update.

```
PROGRAM  myodbr

USE YOMDB
#include "tsmbkind.h

REAL_B :: ZINFO
INTEGER :: INFO(32), INBOBS, JOBS

CALL OPENDB (’ECMA’,’READONLY’,INFO,0)

CALL GETDB(‘MYCONTEXT',0,INBOBS,INFO,0,ZINFO,0,-1,-1,-1,-1,-1)

DO JOBS=1,INBOBS
    WRITE(*,*) ROBHDR(JOBS,MDBLAT), ROBHDR(JOBS,MDBLON)
ENDO

CALL PUTDB('MYCONTEXT’',0,INBOBS,INFO,0,ZINFO,0)

CALL CLOSEDB(.FALSE.)

STOP
END PROGRAM myodbr
```

Opens an ECMA type database for only reading

The number of rows in the query result (i.e. the number of AIREP reports)

Extracts data from the database via context MYCONTEXT into ROBHDR

MDBLAT and MDBLON can be used as indexes for lat@hdr and lon@hdr

Closes database without updating (no update was asked in the context)

In the execution script the location of the input ODB is given via the ODB_SRCPATH_ECMA and ODB_DATAPATH_ECMA environmental variables. The IOASSIGN environmental variable also must be set. For the database type (ECMA) is explicitly specified in the argument of OPENDB it is not necessary to set environmental variable ODB_CMA. See next chapter for details on these environmental variables.

9. Some ODB related programs

The basic aim of this chapter is to give a short overview about programs that I have worked with. Not all the features of these programs are listed here (unfortunately I do not know them) so further readings and investigations are needed to get more details.

The common feature of most ODB related programs is the usage of shell environmental variables to locate the database and to specify some other information. The most important environmental variables are as follows:

- **ODB_CMA**: Specifies the database type (ECMA, CCMA etc.)
- **ODB_SCRPATH_***dbtype*: Specifies the directory of the *dbtype*.dd and *dbtype*.sch files
- **ODB_DATAPATH_***dbtype*: Specifies the directory of pools
- **IOASSIGN**: Specifies the location of the so-called ioassign file used for database description. By default this file is: ODB_sys/cy24t1/vpp5000/*dbtype/dbtype*.IOASSIGN

9.1. ODBTOOLS

odbtools is the basic ODB related program. It can be found in directory ODB_sys/cy24t1/vpp5000/bin as odbtools.x and. This program can work in several modes, which can be specified via the name of the program. The different modes/names are: **to_odb**, **to_ecma**, **shuffle**. The sources of odbtools can be found in ODB_sys/cma2odb. In directory ODB_sys/scripts there is an example script for each of these programs. These scripts are: cma2odb for to_odb, odb2ecma for to_ecma and odbshuffle for shuffle.

9.1.1. odbtools: to_odb

This program creates an ODB database from a CMA-type file. The database can be ECMA, ECMASCR or CCMA. The directory containing the database (specified by ODB_DATAPATH_*dbtype* and ODB_SRCPATH_*dbtype*) must be created in advance, before running to_odb. The argument of the program is the input CMA-type file.

9.1.2. odbtools: to_ecma

This program creates a CMA-type file from an ECMA- or ECMASCR-type ODB database. The result of the program is one CMA file for each pool. The only argument for the program is the prefix of the name of the output CMA-type file(s).

9.1.3. odbtools: shuffle

Shuffle extracts data of a given type ODB database in a certain way and writes them into another type ODB database (it will be created by the program if it is necessary). The distribution of data into the desired number of pools and timeslots is also ensured. If the output database does not exist its directory(ies) must be created in advance, before running shuffle. Shuffle can perform three different database conversions:

- **ECMASCR→ ECMA**: In this case shuffle creates an ECMA-type database with all the data in the ECMASCR-type database. This conversion should be used before **screening** to create an input ECMA-type database with the given number of pools.
- **ECMA→CCMA**: In this case shuffle creates a CCMA-type database using only the active data in the ECMA-type. This conversion should be used after screening to create an input CCMA-type database for **minimisation**.
- **CCMA→ECMA**: In this case the ECMA database must exist and only the minimisation results in CCMA will be added to ECMA. So this conversion should be used after the 3D/4DVAR analysis for **matchup**.

Argument list of shuffle consists of several flags:

-i*idbtype* **-o***odbtype* **-a***ipn* **-b***fp* **-B***andate* **-L***tstartdep* **-R***tenddep* **-T***tsnum* **-F***ftr* **-M**

where:

- *idb*: input database type
- *odb*: output database type
- *ipn*: number of pools in the input database
- *fp*: multiplication factor for the number of pools. The output database will contain *ipn***fp* pools.
- *andate*: analysis date in yyyymmddhh format.
- *tstartdep*, *tenddep*: the departure between the edges of the assimilation time window and the analysis date (in minutes)
- *tsnum*: number of timeslots in the output database
- *ftr*: database type used in final trajectory in minimisation (ECMA or CCMA). It is an optional flag for the matchup.

and **-M** is an optional argument; if it is given then the input database will be opened in READONLY mode.

9.2. Creation of LAM-ODB

For ALADIN 3DVAR screening the input ECMA-type ODB should contain reports only in the C+I zone (thus, this ODB is called LAM-ODB). At present, the creation of a LAM-ODB is done in two steps. In the first step program **lamflag_odb** scans through the whole ODB and sets column subtype@hdr to 1 for reports inside the C+I zone (and to 0 otherwise). Then in the second step (with the modified version of program **shuffle**) only reports with subtype@hdr=1 are selected and written out into a new ODB, which is now suitable for ALADIN purposes. To see these programs in work please, see kami:~mrpm624/devAL15/example/obs2lamodb.run. The documentation of lamflag_odb can be found on kami as ~mrpm624/odb/lamflag/doc/lamflag.doc (or ps).

10. Biblography

1. Puech D.: ODB Guide pratique. Users guide to ODB.
2. Saarinen S.: ODB – Observational DataBase. Presentation for Meteo France, 18-22/09/2000