Version

3.00

Treehouse Software, Inc.

ADAREORG User Guide

A Product of CCA Software Pty Ltd

Copyright Notice

Trademark Acknowledgments

ADABAS and NATURAL are trademarks of SOFTWARE AG of Germany and North America. Solaris is a product of Sun Microsystems and HPUX is a product of Hewlett Packard Corporation of the USA. Any other trademarks referred to herein are the property of their respective owners.

Requirements for Confidentiality

This document contains trade secrets and proprietary information of CCA Software Pty Ltd. Reproduction of this document without the written approval of CCA Software Pty Ltd is prohibited. Use of this document is limited to licensed users of ADAREORG or those given specific written permission of CCA Software Pty Ltd.

THE SOFTWARE WHICH IS DESCRIBED IN THIS DOCUMENT IS SUBJECT TO LIMITATIONS ON USE, RELEASE, DISCLOSURE AND DUPLICATION AND TO REQUIREMENTS FOR CONFIDENTIALITY, PROTECTION AND SECURITY WHICH ARE SET OUT IN THE SOFTWARE LICENCE AND MAINTENANCE AGREEMENTS.

Treehouse Software, Inc.
2605 Nicholson Road, Suite 230
Sewickley, PA 15143 USA
Phone: 724.759.7070
Fax: 724.759.7067
e-mail: tsi@treehouse.com
http://www.treehouse.com

Table of Contents

Chapter 1

INTRODUCTION

Functional Overview

ADAREORG reads ADABAS compressed files such as the ULDDTA output by ADAULD or the CMPDTA output by ADACMP. It then produces new compressed DTA and DVT files, with the same data but with different field layout and definitions according to the needs of the user.

ADAREORG is used by a system administrator to implement design changes in development or production ADABAS files and to prepare test data. Fields may be added, removed, renamed, moved, made empty, given a different format, length, compression attribute (NU, FI), periodicity attribute (MU, PE), or have descriptor status added, removed or changed. Records to be output can be selected according to Boolean combinations of field values. Throughout this change, ADAREORG faithfully and automatically converts the existing data values in selected records to appropriate values in the new output compressed records. The output can then be loaded into a different ADABAS database.

The main input which drives the ADAREORG field reorganization is a "dummy" or "model" compressed file (with zero records) which is set up with an FDT as desired. ADAREORG creates an identically structured output file, filling it with converted data from the input file according to matching field names and any overriding parameters from standard input.

Parameters in standard input are in similar format to other ADABAS utility input. ADAREORG can be run interactively from a terminal but would usually be executed from a script, with input redirected to come from the script itself, and output redirected to a file.

ADABAS Version Compatibility

ADAREORG V2.03 includes support for ADABAS V3.1.x, V5.x & V6.x. ADAREORG V2.03 will only support the basic feature set of V5 & V6 ADABAS releases, ie PE/MU < 191, no BLOBs etc.

Operating system support is for SOLARIS 2.6, 7, 8 and 9, HP/UX 10, 11i, AIX, Linux under i386 (Redhat 8.0 & Advanced Server 2.1), Windows XP and Server 2000/2003. A Specific binary executable is required for each operating system.

To distinguish between versions of ADAREORG, the version number is printed on the ADAREORG Execution Report page header.

Large file support (i.e. > 2GB) is only available on operating systems that support it, e.g. Solaris 7 and the specific level of ADABAS.

Chapter 2

PROCESSING OVERVIEW

Procedure Flow

ADAREORG input sources:

- control statements (**stdin**)
- model - compressed DTA (REODTAM)
- input compressed DTA (REODTAI)

ADAREORG output:

- messages (**stdout**)
- output compressed DTA (REODTAO)
- output compressed DVT (REODVTO)
- error messages (**reoerr**) specifically from ADAREORG, usually same as stdout
- error messages from system (**stderr**), usually same as stdout

Refer to Figure 1 on Page 6, for an operational overview.

Data Set	Environment Variable	Storage Medium	Additional Information
Model compressed	REODTAM	disk	output of ADACMP, ULD (0 records)
Compressed data in	REODTAI	disk	output of ADACMP, ULD
Compressed data out	REODTAO	disk	suitable for ADAMUP input
Compressed descriptors	REODVTO	disk	suitable for ADAMUP input
Control statements	**stdin**		
ADAREORG messages	**stdout**		

Input Data

REODTAI
Primary input is the compressed data (DTA) file that is to be reorganized. This is usually produced by unloading a file from the database via ADAULD, although a DTA from any other utility (including ADAREORG) is acceptable. Note: that the descriptor value (DVT) is not needed for input, as ADAREORG directly builds the output DVT. The SHORT option could therefore be in ADAULD to create the DTA. However, this is not recommended as the compressed input will then have no value as an emergency backup, as it can't be reloaded easily by ADAMUP. The input data filename can be defined to ADAREORG either through the REODTAI environment variable or through the REODTAI input parameter.

REODTAM
Secondary input, which must be present, is the model DTA file. This model file has exactly the same structure as your reorganized output file but has zero records. Its purpose is to tell ADAREORG "by example" how to automatically reorganize the input. ADAREORG compares the model and input FDTs for default conversion action, although the default can be altered by control statements from **stdin** as required. Normally this file would be created first taking a copy of the FDT of the input file (e.g. from PREDICT or ADADCU with numrec=0 and using the DCUFDT option). The FDT is then modified FDT as required, and used in the ADAFDU input to define a new file to ADABAS. This new file can then be unloaded with ADAULD to produce the required model file. The model data filename is defined to ADAREORG either through the REODTAM environment variable or through the REODTAM input parameter.

Stdin

Auxiliary input is the standard input for ADAREORG control statements.

Output Data

REODTAO/ REODVTO

New, reorganized DTA and DVT files are written. These files are ready to load using the ADAMUP utility. The output filenames are defined to ADAREORG either through the REODTAO and REODVTO environment variables or through the REODTAO and REODVTO input parameters.

Stdout

Error messages and file statistics are written to standard output. This output file also contains any messages written to the ADAREORG error file (reoerr) and the system error file (stderr).

End of Processing Messages
At the end of processing, the following is printed:

```
ADAREORG records read:
ADAREORG records skipped:
ADAREORG records rejected:
ADAREORG records written:
ADAREORG max DTA record length written:
ADAREORG max DVT record length written:
```

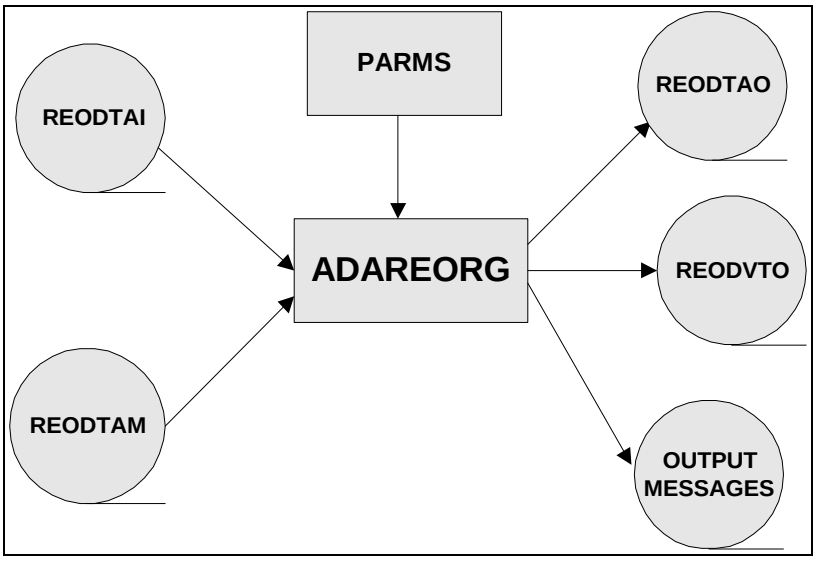

Figure 1: ADAREORG operation

Processing Considerations

ADAREORG will not generate hyper-descriptor or phonetic descriptor values. If these types of descriptors are required it is a simple matter to run the invert function of ADADBS to create them.

ADAREORG will not handle the rarely used "virtual field" facility of ADABAS.

Note: that certain data conversions that ADAREORG may do are not reversible. For example, conversion to floating point (where possible loss of precision may occur), conversions involving truncation, conversions to numeric from alphanumeric (resulting in loss of leading blanks) etc.

Warning messages should be carefully examined. Failure to do so may lead to incorrect data being loaded into the database. Such issues as:

⇒ Warnings about over-length data. The system administrator (sysadmin/DBA) may wish to use a MODE parameter (see above) to enforce truncation, as the default is not to truncate. The truncation mode will only apply to fields that have changed their name, length or format - other fields will be passed through, retaining their over-length characteristic. Normal practice is to not allow fields to be given data lengths greater than the standard length.

⇒ Fixed length fields may attract a field truncation warning if derived from a source field of greater length. Remember that truncation of numeric fields results in loss of high order digits, which should be carefully investigated as this will lead to incorrect values in target fields and should not normally be tolerated.

⇒ Certain conversions (e.g. alpha to numeric, where the alpha contains more than 28 digits) will generate a message to the effect that ADAREORG is "incapable" of the conversion - the result will be a null value in the affected field.

⇒ As field warning messages may "flood" the output, ADAREORG imposes a limit for each kind of message against each field (with a warning message if the limit is exceeded). Additional error messages for the field will be ignored unless they are of an error type which has so far not appeared for that field - in this case, the error message will be output (but only once).

Although ADAREORG has been exhaustively tested, it is important that the System Administrator test the validity of output data prior to loading that file to the database, particularly if complicated or unusual field changes have been made. ADAVFY is one way of obtaining reassurance that a file is internally consistent.

Restart Considerations

If an ADAREORG run fails or is stopped during execution, it must be restarted from the beginning after deleting the output files. It does not matter whether ADABAS is active or inactive; ADAREORG is a stand alone utility that makes no changes to the database and does not use the ADABAS nucleus.

Chapter 3

OPERATIONS

File Definitions

General Syntax Rules
Descriptions of input and output files must use standard ADACMP syntax.

Super Descriptor
Super/Sub/Phonetic-descriptor field definitions can be present but are ignored by ADAREORG.

Level Numbers
ADAREORG deals only with elementary fields. Group fields are ignored.

PE/MU
PE fields are accepted. Members of a PE must have levels greater than 1. Note that PE or MU occurrence counts should NOT be used on ADACMP cards. In keeping with standard ADACMP definitions, PE's may not be nested, but may contain MU's.

Standard Length
This must be present. ADAREORG does not support undefined lengths.

Standard Format
Currently this must be A, U, B, P, or F. The G (floating point) field type is partially supported - the fields are internally translated to B format by ADAREORG. For floating point see new definition on page ##.

Definition Options
The standard field description definition options may be present, i.e. NU, FI, DE, UQ, MU and PE. However, only MU and PE are used by ADAREORG.

Input Commands and Control Parameters

The following control parameters are available (descriptions follow):

```
D  REODTAM = path name
D  REODTAI = path name
D  REODTAO = path name
D  REODVTO = path name
D  DBID = number
D  FILE = number
D  SKIPREC = number
D  NUMREC = number
D  LET target_field = source_field
D  SELECT field_relation AND field_relation AND ...
D  MODE conversion_option
```

As well as the control parameters listed above, there are several other input commands which are mainly used interactively:

HELP or ?	displays a help panel giving parameter summaries.
SHOW or *	displays current parameter values.
!	followed by a command executes a shell command.
QUIT	abandons the interactive session.
START	terminates the parameter phase and initiates execution.

Control Parameters

ADAREORG keywords are case insensitive.

REODTAM

```
REODTAM = path name
```

Optionally override any REODTAM environment variable. It must be the file name of a compressed DTA file produced by ADAULD or other utility. It can be "empty" because it is only read to obtain the new FDT details.

REODTAI

```
REODTAI = path name
```

Optionally override any REODTAI environment variable. It must be the file name of a compressed DTA file produced by ADAULD or other utility.

REODTAO

```
REODTAO = path name
```

Optionally override any REODTAO environment variable. It must be the file name for the compressed reorganized data.

REODVTO

```
REODVTO = path name
```

Optionally override any REODVTO environment variable. It must be the file name for the compressed reorganized associator values.

DBID

```
DBID = number
```

This is the target (output) database id, which will be embedded in the DTA and DVT headers. Default is the input database id from ADAULD.

FILE

```
FILE = number
```

This is the target (output) file number, which will be embedded in the DTA and DVT headers. Default is the input file number from ADAULD.

SKIPREC

```
SKIPREC = number
```

This specifies the number of records (i.e. ISNs) that will be skipped before writing records to the output FILE.

NUMREC

```
NUMREC = number
```

This specifies the maximum number of records (i.e. ISNs) which will be written to the output FILE.

LET

```
LET target_field = source_field
LET target_field = EMPTY
```

This causes the output **target_field** to be built from data in the input **source_field**.

The target and source fields are ADABAS 2-byte field names.

If there is no LET parameter applicable to an output field, then its value is taken from the field of the same name in the input (with appropriate conversion for format differences) or set to null if no corresponding field exists.

The EMPTY variant of this statement forces the output field value to be null.

LICENCE

```
LICENCE = codeword
```

This is a mandatory parameter, the "codeword" is provided by your local distributor, which will enable the product to run on the designated (and therefore licensed) machine.

SELECT

```
SELECT field_relation AND field_relation AND...
```

This causes a record to be selected when ALL the **field_relation** expressions are true. There can be more than one SELECT statement present, in which case the different SELECT statements are linked by an implicit logical OR.

'field_relation' is of the form:
left_operand <space> **relational_operator** <space> **right_operand**

e.g. AA < 5.

The left operand must be an ADABAS field name present in the input DTA file, which must be a simple field (i.e. not an MU or member of a PE group). The relational operator must be one of <, >, <=, >=, =, <>. The right operand can be another non-PE, non-MU ADABAS input field name, or it can be a constant. Constants can be quoted alphanumeric (e.g. 'part01'), numeric (e.g. 99), or hexadecimal (e.g. x'01020aff') as appropriate - depending on the format of the field named in the left operand.

MODE

```
MODE  LEADING ZERO
MODE  NOT LEADING ZERO
```

Default is **not leading zero**.
The **leading zero** mode affects field data conversions from numeric to alpha format. Leading zeros are added to the number to make it the standard length of an alpha (A) field after conversion.

```
MODE  BLANK WHEN ZERO
MODE  NOT BLANK WHEN ZERO
```

Default is **not blank when zero**.
The **blank when zero** mode affects field data conversions from numeric to alpha format. When the numeric source value is zero, it is converted to a blank (which may be null suppressed) rather than a single zero digit.

```
MODE  NUMERIC TRUNCATION
MODE  NOT NUMERIC TRUNCATION
```

Default is **not numeric truncation.**
The **numeric truncation** mode is effective only when an output numeric field (P, U, F, or G format) generated by ADAREORG conversion would exceed the standard length of that field. Normally an output field that is over standard length (but still a legal length), will be built to the longer length, and a warning message generated. However, if the source and target fields have the same name, format and length, then the data is copied as is. The numeric truncation mode suppresses any warning messages, and builds a standard length output field, truncating the leftmost digits to fit.

```
MODE  ALPHA TRUNCATION
MODE  NOT ALPHA TRUNCATION
```

Default is **not alpha truncation**.
The **alpha truncation** mode is effective only when an output alphanumeric field (A format) generated by ADAREORG conversion would exceed the standard length of that field. Normally an output field that is over standard length (but still a legal length) will be built to the longer length, and a warning message generated. However, if the source and target fields have the same name, format and length, then the data is copied as is. The alpha truncation mode suppresses any warning messages, and builds a standard length output field, truncating the rightmost bytes to fit.

```
MODE  NUMERIC OVERLENGTH
MODE  NOT NUMERIC OVERLENGTH
```

Default is **not numeric overlength**.
The **numeric overlength** mode is effective only when an output numeric field (P, U, F, or G format) generated by ADAREORG conversion would exceed the standard length of that field. Normally an output field that is over standard length (but still a legal length) will be built to the longer length, and a warning message generated. However, if the source and target fields have the same name, format and length, then the data is copied as is. The numeric overlength mode suppresses any warning messages.

```
MODE  ALPHA OVERLENGTH
MODE  NOT ALPHA OVERLENGTH
```

Default is **not alpha overlength**.
The **alpha overlength** mode is effective only when an output alphanumeric field (A format) generated by ADAREORG conversion would exceed the standard length of that field. Normally an output field that is over standard length (but still a legal length) will be built to the longer length, and a warning message generated. However, if the source and target fields have the same name, format and length, then the data is copied as is. The alpha overlength mode suppresses any warning messages.

```
MODE  NUMERIC BINARY
MODE  NOT NUMERIC BINARY
```

Default is **not numeric binary**.
The **numeric binary** mode is effective only for conversions from unsigned binary (B format) to alphanumeric (A format). The effect of this mode is that the B field (which must have a length of 4 bytes or less) will be interpreted as a pure number and will be converted into ASCII digits. If this mode is not in effect, up to 126 bytes are moved untranslated, but leading binary zeros and trailing spaces (i.e. binary 32) are dropped.

Parameter Examples

```
#= indicates a comment line
REODTAM=/adabas/model/f002.DTA
REODTAI=/adabas/f001.DTA
REODTAO=/adabas/f002.DTA
REODVTO=/adabas/f002.DVT
   FILE = 59             ;control parameter lines can be freely indented
   SKIPREC = 20
   NUMREC = 999
LET AA = EMPTY
LET AB = Z1
SELECT AA = 'central' AND AB <> 0
SELECT AA = 'district' AND AB >= 100  ; will be ORed with line above
MODE LEADING ZERO
MODE BLANK WHEN ZERO
```

Sample Case Study

You are a DBA in a large manufacturing organization that maintains an ADABAS product file. Changes in the manufacturing area necessitate an expansion in product codes and product descriptions. The Product Code is an 8-byte U format field - it needs to be changed to an 8-byte A format field in order to hold alphanumeric codes for the new product range. In addition, a new field needs to be added to the Product Description (which is held in a PE group). The data in the existing file must to be converted to match the new requirements. This is a job for ADAREORG.

Firstly, take a standard unload of the product file using the ADAULD utility. Then a copy of the FDT for that product file is needed. Since only the FDT is needed, the ADADCU utility is run against the compressed data just unloaded, using the DCUFDT option and the NUMREC=0 option – producing the desired result.

The next step is to edit the FDT to reflect the needed changes. In our example we change the format of the Product Code field from U to A and insert a new field definition as required into the Product Description PE group. While looking at the FDT, it is decided to delete two fields that were made obsolete a year ago and similarly an obsolete descriptor. It is also decided that a number of fields should have been null suppressed, so those fields are changed with the editor to add NU option against them in the FDT.

Now you want to create a test file (with the new FDT structure) in your development database so that NATURAL programmers can make changes to associated programs. You run ADAFDU on the development database, with appropriate DSSIZE parameters etc. and with the FDUFDT environment variable set to the filename of the FDT that you created in the previous step.

The result of this ADAFDU is an empty product file with the desired new structure. You will later load this with new records created by ADAREORG from the original unloaded product file data.

However, ADAREORG needs to be "told" how to reorganize the data from the unloaded product file, i.e. it needs a revised "model" FDT to follow. This is achieved by running ADAULD to unload a newly defined **empty** product file from the development database, containing a suitable model FDT in its header information.

Now you set up a script to run ADAREORG with your two compressed (ADAULD format) files as input. The large compressed ULDDTA file from the production product file is defined to ADAREORG via the REODTAI environment variable ("I" for input). The small compressed ULDDTA file from the newly defined development product file is defined to ADAREORG via the REODTAM environment variable ("M" for model).

To complicate matters further, you are working under a space limitation in your development database (into which you plan to load the ADAREORG output). Therefore, you decide to apply some ADAREORG selection criteria in order to reduce the size of the output test file. Having decided that products originating from the Sydney factory in 1994 or later would make a suitable subset for testing, you add the following line to your ADAREORG parameters:
(Where field FL is a factory location code and field DA is a date).

```
SELECT FL = 'SYD' AND DA > 1994000
```

Before you actually run the script, you decide to talk to the application system designers to find out what should be done with zero value product codes as you are concerned about how ADAREORG will convert them to alpha. You discover that there are indeed product codes with a zero value and as the field is null suppressed and a descriptor, it is very important that they are represented as a null rather than '0'. You use the ADAREORG parameter

```
MODE  BLANK WHEN ZERO
```

to ensure that this will happen. The designers also specify that they would prefer that leading zeros be added to short product codes ('00009950' instead of '9950') therefore use the following parameter:

```
MODE  LEADING ZERO
```

Having specified all required parameters, ADAREORG is run by executing the script set up previously - resulting in the creation of the files specified by environment variables REODTAO and REODVTO.

The final step is to load the reorganized files into the development database. This is achieved via ADAMUP (with the ADAREORG output files now named in the ADAMUP environment variables MUPDTA and MUPDVT respectively). The ADAMUP utility is run and the job is complete. ADAREORG has automatically converted the data in the changed fields so it conforms to the new description, has added null data for the field you added in the PE group, and has removed all traces of data for the obsolete fields that were removed from the FDT.

After the programmers have made their changes and the system has been thoroughly tested in development, the DBA is then required to migrate the programs into production and adjust the production database to suit. Simply repeat the entire process as used above - but this time making your production database the ultimate target, and omitting the SELECT parameter used to filter the test database.

Note that the original product file (in the production database) must be deleted prior to running the script.

As a precaution, it is recommended to check that the data and programs are good, i.e. undertake a standard set of system tests for this application prior to general release. Sources of information that may be used to assist are record counts, unusual utility messages, comparative file sizes for "reasonableness", and run some batch reports.

Note – The DBA may choose to run the ADAVFY utility on the new file, or even consider running SHOWCMP (a utility which comes with ADAREORG which can "dump" a compressed file showing field by field contents) if for some reason the data conversion is not as expected. The DBA should always be prepared to back-out if necessary, by de-installing the new NATURAL programs and reloading the production file from the compressed file (used as ADAREORG input), in the unlikely event that this should be needed.

After the testing process has been successfully completed, take the opportunity to set up some standard scripts and procedures for the local site, which "automate" most of the process of "reorganizing" a file. This will make the whole process a standard operational procedure when a file or files need to be structurally modified in the future. The benefit of this process is that it enables the system administrator (or DBA) to then rummage through the "too hard" basket to see if there are any other outstanding maintenance requests, which could be handled by ADAREORG. As part of the implementation of ADAREORG it is a good idea to inform application development teams of the ease with which file design changes in systems under construction, or in fact existing systems, can now be made, although it should not be used to cover for less analysis in a given project.

Some Sample Scripts

The following two skeletal scripts indicate how a DBA might approach the task of modifying file 20 in database 1 using ADAREORG. There are 3 steps in script 1 and 5 steps in script 2, but most of these steps are trivial (i.e. consume a very small amount of machine resources) except for the initial unload, the ADAREORG execution and the reload.

```
#!/bin/sh
# script 1, step 1 (unload existing file)
#
LICENCE=BIPJTNPHHJJJJHMKIHKH
ULDDTA=original.DTA
ULDDVT=original.DVT
export ULDDTA ULDDVT
adauld << END_OF_PARAMS
db=1
file=20
END_OF_PARAMS
#
# script 1, step 2 (obtain an fdt for the existing file)
#
DCUDTA=original.DTA
DCUFDT=original.fdt
export DCUDTA DCUFDT
adadcu << END_OF_PARAMS
numrec=0
dcufdt
END_OF_PARAMS
#
# script 1, step 3 (make a copy of the fdt for editing)
#
cp original.fdt new.fdt
#
#------ end of script 1 ---------------
```

After running script 1, you must check its output and manually edit the FDT (new.fdt) to make the field modifications you desire. You then run script2 (below) after ensuring that the filenames and numbers are correct and that the ADAFDU and ADAREORG parameters are appropriate for the new file.

```
#!/bin/sh
# script 2, step 1 (delete original file)
#
LICENCE= BIPJTNPHHJJJJHMKIHKH
adadbm << END_OF_PARAMS
dbid=1
delete=20
END_OF_PARAMS
#
#
# script 2, step 2 (define new file to ADABAS)
#
FDUFDT=new.fdt
export FDUFDT
adafdu << END_OF_PARAMS
dbid=1
file=20
  ----------------------------------------------------------------
       here insert parameters for MAXISN, ISNSIZE,
       NAME, ASSOPFAC, DATAPFAC, UISIZE, NISIZE,
       DSSIZE, REUSE and other ADAFDU options.
  ----------------------------------------------------------------
#
END_OF_PARAMS
#
#
# script 2, step 3 (generate new model file for ADAREORG)
#
ULDDTA=model.DTA
export ULDDTA
adauld << END_OF_PARAMS
db=1
file=20
short
END_OF_PARAMS
#
```

(continued over page)

```
# script 2, step 4 (convert compressed files with ADAREORG)
#
LICENCE= BIPJTNPHHJJJJHMKIHKH
#
REODTAI=original.DTA
REODTAM=model.DTA
REODTAO=new.DTA
REODVT=new.DVT
export REODTAI REODTAM REODTAO REODVTO
adareorg
#
# << END_OF_PARAMS
db=1
file=20
END_OF_PARAMS
#
#
# script2, step 5 (load the new files into ADABAS)
#
MUPDTA=new.DTA
MUPDVT=new.DVT
export MUPDTA MUPDVT
adamup << END_OF_PARAMS
db=1
update=20
add
END_OF_PARAMS
#
```

To develop tailored scripts for ADAREORG along the lines of this example, some additional information is required. Prior to running the script the ADABAS environment must be setup - most sites run a script called "adaenv" then define and export DATAB and run the SAG supplied "database.bsh" - this needs to be done at the beginning of each script as it sets up environment variables which enable the ADABAS utilities to know how to access the database. The ADABAS environment setup will be the same as in the local ADABAS utility scripts.

Filenames should be defined in accord with the local site own standards. It is recommended that extra statements be added to the script to echo return values. The script could be parameterized so that **dbid** and **file** are variables (so that they need only be changed in one place). An optional step may be added at the end to run ADAREP or ADAVFY on the new file.

It is recommended that a third script be created for back-out purposes. This script would contain steps to delete the file, redefine it as it originally was, and reload it from the original DTA and DVT files (or alternatively recover it from an ADABCK backup). This safety step would normally be the case with any production change and is not specific to ADAREORG.

Chapter 4

FIELD FORMAT CONVERSIONS

Length Conversion

When a conversion results in a field length which is longer than the standard length of the field (but which is still a legal length), a warning message is usually generated. The exception is when "long alpha" or "truncate alpha" is specified (for A fields) or "long numeric" or "truncate numeric" is specified (for non-A format fields).

A special exception is "straight through" fields, which derive from a source field with the same name, format and length - these are simply passed to the output file as overlength.

If none of the above is specified, the field is passed as overlength and a warning message is generated, unless FI (FIXED), in which case it is truncated.

If "long" appropriate to the format is set, the overlength will be passed and the warning message suppressed.

If "truncate" appropriate to the format is set, the overlength will be truncated (on the right for alpha and left otherwise) to the standard length and the warning message suppressed.

Note that where several settings conflict, the resolution is as follows: "straight through" fields are never truncated or given overlength warnings, otherwise "truncate" setting is the next priority, and "long" is the lowest priority.

Data Type Conversions

Warning - Data type conversions should be undertaken with care, as some results are non-reversible, e.g. when truncation of data occurs. Always make a backup copy of the input file prior to undertaking data type conversion.

Special Note - An explanation of the terminology used to describe data type conversion results, is as follows:

⇒ A "**simple conversion**" is defined as one where the field type is changed to reflect the desired result, but the field contents remain essentially unaltered in any other way. This can only be achieved in instances where the source format and the target format are compatible, e.g. "alpha-to-binary" or "packed to unpacked".

⇒ A "**complex conversion**" is defined as one where the source format and the target format are incompatible and field contents may therefore be physically altered in some way, e.g. "alpha-to-numeric". Complex conversions can result in truncation or other necessary modification to data.

From ALPHA (type A)

Alpha-to-binary is treated as a simple conversion.
Alpha-to-numeric (of any type) is a complex conversion. Any trailing blanks or excess data beyond 32 characters is dropped, non-numeric characters are translated to zeros, and the result is then treated as if it were unpacked.

From UNPACKED (type U)

As Unpacked fields are essentially compatible with any other type, all conversions from Unpacked are treated as simple conversions.
Note - up to 27 signed digits can be accommodated in a U type field.

From BINARY (type B)

Binary-to-alpha is treated as a simple conversion.
Binary-to-fullword is treated as a complex conversion. If the length is four bytes or less, and MODE NUMERIC BINARY is specified, it is assumed that the B field contains a fullword, and it is treated as such. If the length is greater than 4 bytes, and MODE NUMERIC BINARY is specified, an error will result. If the length is greater than 4 bytes, and MODE NUMERIC BINARY is NOT specified, then the field is simply copied (based on the assumption that the binary field already contains data formatted according to the definition of the output field).

From FULLWORD (type F)

Fullword to either packed or unpacked is treated as a simple conversion.
Fullword to binary or alpha is also treated as a simple conversion.

From PACKED (type P)

Packed-to-alpha is treated as a complex conversion.
Packed-to-binary is treated as a simple conversion.
Packed-to-unpacked or any fullword numeric form is treated as a simple conversion.
Note - up to 27 digits of signed packed decimal (14 bytes) can be accommodated in type P field.

From Floating Point (type G)

This is treated as a complex conversion.
An attempt is made to retain the numeric value of the converted field, however, as floating point numbers can far exceed the capacity of other formats, it is recommended to carefully check these conversions.

Periodicity (PE/MU) Conversion

The following rules apply when manipulating Periodic fields (PE) and/or Multi-valued fields (MU). Please take careful note of field processing carried out by ADAREORG as data may be truncated in "periodicity" conversions.

There are four types of source field and four types of target field, namely, SIMPLE (S), PE (P), MU (M) and PEMU (X).

Let SS be a source field and TT be a target field.

Let AAn be nth occurrence of P field AA.

Let AA(n) be nth occurrence of M field AA.

Let AAi(j) be the jth occurrence of the MU field AA which is found in the ith repetition of a PE.

Simple fields:

S to S:	TT = SS.	
S to P:	TT1 = SS	(possibly null, TTi where i = 2 to PE max, are null values.)
S to M:	TT(1) = SS	(possibly null, if NU then whole field replaced by empty field count.)
S to X:	TT1(1) = SS	(possibly null, all other TT values are null.)

PE fields:

P to S:	TT = SS1	(if it exists, else null.)
P to P:	TTi = Ssi	(where i = 1 to TT PE max, if it exists, else null.)
P to M:	TT(i) = Ssi	(where i = 1 to SS PE max.)
P to X:	TTi(1) = Ssi	(where i = 1 to TT PE max, if it exists, else null.)

MU fields:

M to S:	TT = SS(1)	(if it exists, else null.)
M to P:	TTi = SS(i)	(where i = 1 to TT PE max, if it exists, else null.)

M to M:	TT(i) = SS(i)	(where i = 1 to SS MU max.)
M to X:	TT1(i) = SS(i)	(where i = 1 to SS MU max.)

PEMU fields:

X to S:	TT = SS1(1)	(if it exists, else null.)
X to P:	TTi = SSi(1)	(where i = 1 to TT PE max, if it exists, else null.)
X to M:	TT(i) = SS1(i)	(where i = 1 to SS MU max, if it exists, else null.)
X to X:	TTi(j) = SSi(j)	(where i = 1 to TT PE max and j = 1 to SS MU max; if it exists, else null.)

Chapter 5

INSTALLATION

Installation of ADAREORG for the command line (batch) interface is a trivial exercise. The procedure is detailed as follows:

1. Decompress the supplied installation files to a folder named ADAREORG on your local machine.

2. The resulting files will consist of a similar set of files as follows:

    a. Two PDF documents, consisting of the release notes and the Users Guide;

    b. A series of ASCII text files, containing example scripts to run ADAREORG and various ADABAS utilities which may be of use, these should be tailored to the local environment. They must be FTP'd to the environment where ADAREORG is to be used;

    c. One or more binary executable files, which will need to be FTP'd to the target environment where ADAREORG is to be used.

3. Once ADAREORG is loaded, it along with all of the scripts will need to be marked as executable prior to any attempt to run them. Further modification will be required to support the local naming conventions for files and databases.

The ADAREORG GUI interface is only available in the Windows NT, 2000p or 2000 Server environments. Additional installation steps are required. At this time the GUI is only available as a beta release and on specific request to CCA.