Adabas Bridge for DL/1

Conversion

Version 2.3.2

November 2016

This document applies to Adabas Bridge for DL/1 Version 2.3.2.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: ADL-CONVERSION-232-20161117

Table of Contents

Conversion

This documentation provides an overview of the Conversion procedure for the Adabas Bridge for DL/I.

The following topics are covered:

1 Introduction

This manual describes the conversion process “DL/I database to one or more Adabas files” as supported by the Adabas Bridge for DL/I (ADL) conversion utilities. The ADL conversion utilities provide a powerful tool for an automated migration from DL/I to Adabas. The stages comprised by the conversion utilities are:

- analyses and conversion of DL/I database definitions (DBDs) and program specification blocks (PSBs) into entries in the ADL directory for later use by the ADL Interfaces;
- conversion of DL/I database definitions (DBDs) into Adabas file descriptions;
- conversion of the actual data from its DL/I format into one or more Adabas files;

Please note that throughout the entire ADL documentation the term “ADL file” is used to designate an Adabas file originating from the conversion of a DL/I database. This is to point out the particular properties of such a file.

Besides the conversion utilities, the further functional units of ADL are the CALLDLI Interface and the Consistency Interface. The CALLDLI Interface allows DL/I applications to access ADL files in the same way as original DL/I databases. The Consistency Interface provides access to ADL files from Natural programs or with Adabas direct calls. This interface preserves the hierarchical structure of the data, which is of importance for ongoing DL/I applications. The installation and operation of the interfaces is covered by the *ADL Interfaces* documentation.

The current manual primarily addresses the system programmer in charge of the database conversion process. However, certain decisions to be taken in preparation of or during the conversion process will need coordination with the Adabas database administrator and Natural/Adabas application programmers.

A familiarity with the operating system in use is presumed in this manual. Also, a certain degree of familiarity with both database systems is required. This manual makes frequent use of terms, synonyms, abbreviations and facts related to these systems. For clarity, a glossary is provided with the *ADL Messages and Codes* documentation.

The current manual applies to both z/OS and z/VSE type operating systems. References valid for only one operating system are clearly marked as such. The term “DL/I” is used as a generic term for IMS/VS and DL/I DOS/VS.

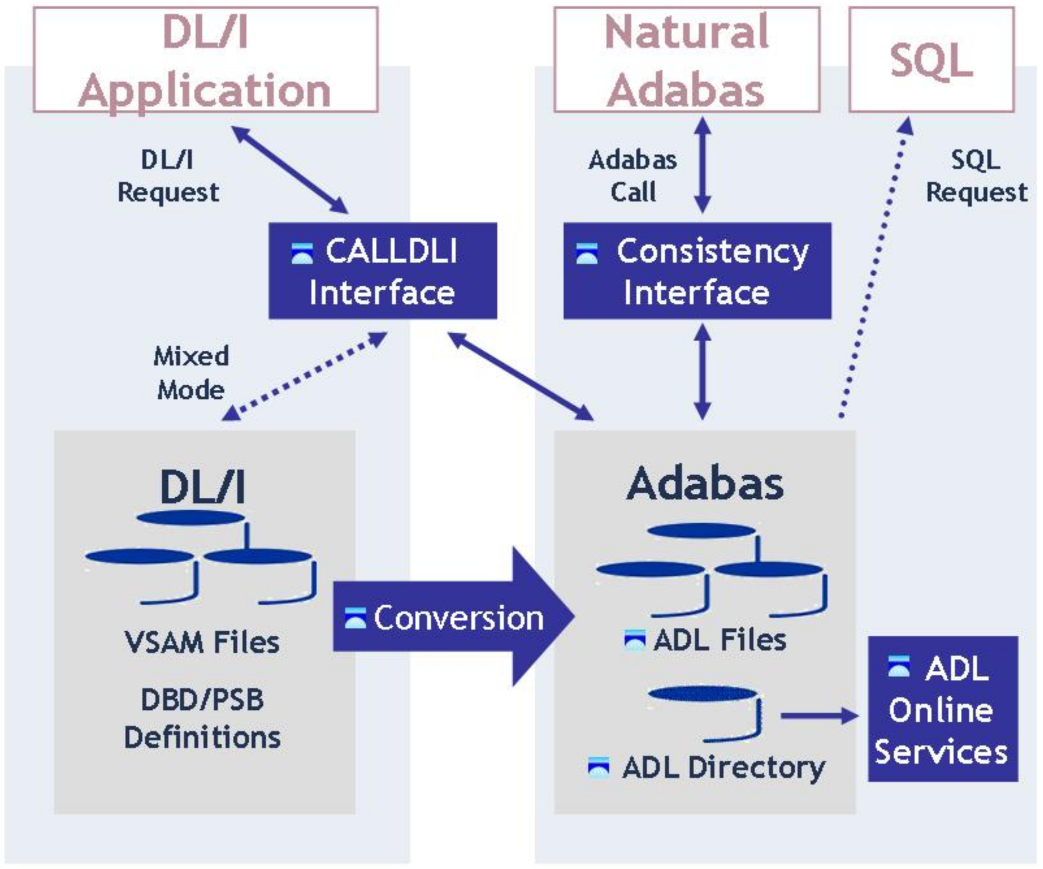


Figure 1: Functional units of ADL and their interrelation

Figure 1 shows the individual functional units of ADL and their interrelation with DL/I, SQL and Natural applications.

This chapter covers the following topics:

Other Documentation You May Need

The following publications may be useful when installing and operating the ADL Interface:

- *Adabas Utilities Manual*
- *Adabas Operations Manual*
- *Adabas Messages and Codes*
- *Adabas Reference Data* and *Adabas DBA Reference Data* documentation.

For a complete list of manuals, prices and ordering information, refer to the **Software AG Documentation Overview** or contact your support representative.

Documentation Related to non-SAG Products

The documentation mentioned below might be of interest and helpful during the ADL conversion process.

For z/OS users:

- IMS/VS Application Programming
- IMS/VS Application Programming for CICS/VS Users
- IMS/VS Utilities Reference Manual
- CICS TS Installation Guide
- CICS TS Operations and Utilities Guide
- CICS Resource Definition Guide

For z/VSE users:

- DL/I DOS/VS Guide for New Users
- DL/I DOS/VS Application Programming: CALL and RQDLI Interface
- DL/I DOS/VS Application Programming: High Level Programming Interface
- DL/I DOS/VS Utilities and Guide for the System Programmer
- DL/I DOS/VS Resource Definition and Utilities
- CICS TS Installation Guide
- CICS TS Operations and Utilities Guide
- CICS Resource Definition Guide

2 Planning the Conversion Process

A successful conversion from DL/I to Adabas presupposes a clear knowledge of the existing system (which DL/I databases exist, which applications access these databases in batch or online mode etc.) and well defined goals. This section outlines some aspects of the conversion process which might be worth considering prior to the conversion.

This chapter covers the following topics:

Stocktaking

The first task in the conversion process is obviously a stocktaking of the existing DL/I databases, PSBs and applications in your environment. The result of this stocktaking process should be a list of DL/I DBDs, PSBs and application programs and should also show which applications use which PSBs and, in turn, which DBDs are referenced by the individual PSBs.

You should be aware that the ADL conversion utilities require the DL/I DBDs and PSBs to be available in source form. The above mentioned list thus might also contain the information where the corresponding DBD or PSB source code can be found.

During the conversion process, you might wish to convert DL/I databases in groups according to their interrelation with applications. On a first level, this could simply be based on whether two DBDs are referenced in one and the same PSB. Also, an online application might schedule several PSBs, which in turn might reference several DBDs. These DBDs could then be considered to be interrelated via this application.

Despite the fact the ADL CALLDLI Interface allows applications to run in mixed mode (accessing original DL/I databases and ADL files concurrently), it might turn out that the conversion process can be simplified and the testing phase shortened if databases related to one and the same application are converted at the same time.

Note that any two physical DBDs which are interrelated by a DL/I logical relationship must be converted together.

Resource Allocation

When large databases are to be converted, it is important to consider the space that will be required for these databases after the conversion. You should be aware that, during the conversion process, the space owned by the database is needed twice, namely in the DL/I system as well as in the Adabas system.

It is recommended that the Adabas database administrator is consulted prior to the conversion process, to determine the space needed for ADL files.

Similarly, the time required for the unload and load batch jobs during the conversion of a DL/I database is comparable to the time needed, for example, for a reorganization of the DL/I database.

3 Conversion of the Data Structure - General Considerations

The operation of the Adabas Bridge for DL/I normally does not need any change of the original DL/I definitions. It is sufficient to convert these original DL/I definitions with the ADL Control Block Conversion (CBC) utilities as described in the section *ADL Conversion Utilities for DBDs and PSBs* in this documentation. The resulting Adabas file layout can be used without any further modification by DL/I or Natural applications through the ADL CALLDLI or ADL Consistency Interface respectively.

Nevertheless, it may be advantageous to depart from this "straightforward" method by introducing some modifications before, during and after the conversion. The section *Managing ADL Files* in the *ADL Interfaces* documentation describes the steps necessary to perform this modification.

The following section offers you details of the default Adabas file layout, as well as some hints on modifying the DL/I structures or the Adabas file layout.

This chapter covers the following topics:

Changes to DL/I Data Structures Prior to the Conversion

Before starting the conversion process, you should check whether any modifications to the DL/I definitions are desired. When the conversion is completed, some further change of the DL/I structures requires a time-consuming unload and reload of the data.

Validating Segment Layouts and Data Types

For DL/I applications, the view of a segment is often not the same as defined in the original DL/I DBD. This is possible, because DL/I corresponds with the application program on a segment level, that means, it offers and accepts a whole segment, regardless of the field definitions. Therefore, a DL/I field can contain non-numeric data, even if it is defined as numeric. Since Adabas operates on a field level, problems can occur while loading such incorrect data.

On the other hand, DL/I fields can exist which are defined as alphanumeric but which contain only numeric data. The CBC utility converts such fields to alphanumeric Adabas fields and Adabas will not validate them for numeric contents. The same effect occurs for parts of a segment which correspond to no DL/I field definition, but which are subsequently used for numeric data.

Therefore, you are recommended to compare the segment layout of the DL/I definitions with the redefinitions (like copy-codes, copy-books) in the application programs. Remove all inconsistencies and include field definitions from the copy- codes into the DL/I DBD definition. You can add all field definitions or combine subsequent alphanumeric fields to form one large field. This is advantageous for the Adabas compression and for the developing of Natural applications as described later.

Note that additional field definitions in the DL/I DBD will not influence your DL/I applications, if the overall segment length is unchanged and you do not define a field as numeric which is supplied with non-numeric values. This is because ADL corresponds with the application program on a segment level like DL/I. But the data is stored with Adabas, and thus you will receive a non-zero response code if an inconsistency in the data type is detected.

Optimization with Respect to Adabas Compression

Once the data is converted, it will be compressed by Adabas as described in the *Adabas Utilities* documentation. Because ADL defines all fields with the “NU” option for the Adabas compression utility ADACMP, the “null value suppression” will also be active.

If the converted data structures do not reflect the structures which are used by the application, the Adabas compression may not operate in an optimal way. Assume for example, there is a segment in a DL/I DBD with no field definitions specified. The application program may redefine this segment by splitting it up into five different parts. If the last part is filled with data, no compression can take place, even if the other four parts are empty.

Another failure of the Adabas compression can be an incorrectly defined field type. If a field contains numeric data but is defined as alphanumeric, it will not be compressed.

Thus, in order to optimize the Adabas compression, the converted data structures and types should reflect the usage in the application programs. This may be achieved by modifying the DL/I definitions prior to the conversion or by changing the Adabas file layout after the conversion.

Adabas File Layout

The ADL Control Block Conversion offers you an automated way to convert DL/I data structures into an Adabas file layout. The following sections outline the default layout and how you can modify it:

- Default Layout Generated by the ADL Conversion Utilities
- Changing the Default Layout

- Considerations for Natural / Adabas applications

Default Layout Generated by the ADL Conversion Utilities

The ADL Control Block Conversion (CBC) converts every element of a DL/I physical DBD into an element of an Adabas file. Additionally some fields are generated to reflect the hierarchical structure. The information about the other DL/I definitions, like PSBs, logical or secondary index DBDs, are kept in the ADL directory file. These structures do not have any influence on the Adabas file layout. The only exceptions are user data fields in index DBDs.

In the following, the translation of the different elements is described in detail:

- A physical DBD corresponds to an Adabas file (an “ADL file”). You may split up the segments of the DBD into different files as described in the next section.
- For every DL/I segment, the CBC utility generates one Adabas group. The overall length of all Adabas fields of this group is the same as the length of the DL/I segment. The name and the overall length of the group must not be changed after the CBC utility runs. There are special rules for logical child segments, which are explained later in this section.
- Every DL/I field is converted to an Adabas field. Redefined or overlapping fields are split up into the smallest units. The CBC utility generates one Adabas field for every unit. The only exception is the root sequence field, which is never split up. For those parts of the DL/I segments which correspond to no DL/I field, so-called Adabas “filler fields” are generated.

An Adabas field has the same type and length as the corresponding DL/I field. Filler fields are always defined as alphanumeric. If an Adabas field would be longer than 253 bytes, it is split up. This arbitrary splitting up may result in problems for developing Natural applications. See the section *Considerations for Natural/Adabas Applications* later in this documentation on how to avoid these problems.

- For every secondary index, one Adabas field is generated as part of the Adabas file, which contains the source segment data. This field is one of the ADL internal fields and must not be altered by Natural applications.
- If an index DBD contains user data fields, then the corresponding Adabas fields are generated additionally to the secondary index field.
- With every ADL file, the ADL physical pointer fields Z0 - Z8 are generated as needed. These internal fields are used by the ADL CALLDLI Interface in order to locate the hierarchical position of a DL/I segment. The contents of these fields must never be altered by Natural applications. The maintenance of all ADL internal fields for Natural applications is performed by the *ADL Consistency Interface*, as described in the *ADL Interfaces* documentation.
- For every child segment type, the ADL CBC utility generates the logical pointer fields (so-called “partial concatenated key fields”, short “PCK fields”) of all parent segments. This concerns the physical as well as the logical hierarchical structure, regardless of which file contains the definition of the corresponding segment group. These fields reflect the hierarchy to Natural

applications. For DL/I applications they are maintained by the ADL CALLDLI Interface automatically.

- The following fields are allocated as Adabas descriptors in an ADL file:
  - the root sequence field,
  - the physical pointer fields Z0 and Z1,
  - the secondary index fields.
  - Logical relationships between two DL/I DBDs require the definition of logical child segments on both sides. The data may be stored on both sides or only on one side. A logical child segment is called "real" if the data is stored on its side; otherwise it is called "virtual".

  The ADL CBC utility will include the elements which correspond to a logical child segment only in one Adabas file, regardless how it was defined in DL/I. ADL will choose the side for which the corresponding logical child segment fulfils one of the following criteria (in order of precedence):

  - The logical child segment has dependent segment types (so-called "variable intersection data").
  - The logical child segment is real, while the paired one is virtual.
  - Both logical child segments are real, and the one in question is to be converted first.

If a logical child segment contains no intersection data (in other words, no data apart from the DL/I concatenated key), no Adabas group for this segment will be generated at all. In fact, all information about this segment is kept in the ADL PCK fields.

Changing the Default Layout

The layout of an ADL file is primarily defined by the DL/I DBD definition. Thus, every modification of the DBD will alter the Adabas file layout.

During the CBC utility run, you can specify the `"FNR"` parameter with the `GENSEG` function. Then the segments will be distributed on different Adabas files. You may also specify the `"ADANAME"` parameter, in order to define the name of the Adabas group, which correspond to a DL/I segment type.

Once the DL/I data structures have been converted to Adabas, some changes are still possible:

- You may define additional fields "inside" or "outside" of segments (i.e. the corresponding groups), provided that the overall length of these groups remains unchanged.
- You may define descriptors, superdescriptors, etc.
- You may alter the Adabas DBID and file numbers of the ADL files by copying the file and reconvert the DBD definitions.

For more information concerning the Adabas layout of an ADL file, see the section *Managing ADL Files* in the *ADL Interfaces* documentation.

Considerations for Natural / Adabas applications

Once the DL/I data structures are converted to Adabas, DDMs can be generated with SYSDDM or with Predict which are used by Natural applications. Because the incorporate functions of these tools use the Adabas layout, you should consider the DL/I structures before the conversion because this defines how the Adabas fields looks like.

As described earlier, a field which would be longer than 253 bytes is split up. This might cause problems for Natural applications, if for example this splitting is in the middle of a numeric value. Thus, you are recommended to force a splitting manually at a better place. This can be performed either by including additional DL/I field definitions into the DBD or by a reorganization of the Adabas fields inside the affected group.

Another reason for avoiding such long Adabas fields is the inflexibility of these fields. As described in the section *Managing ADL Files* in the *ADL Interfaces* documentation, it is easy to increase the size of a field as long as it does not exceed the maximum size of 253 bytes.

You can distribute the segments of a DL/I DBD into different Adabas files. This can be helpful if Natural programs have to identify segment data and segments with no sequence fields are involved. Splitting up the data into different files might also affect the Adabas performance, as described in the section *Performance Considerations* in the *ADL Interfaces* documentation.

If you are using Adabas FASTPATH and some of the segments in a DBD contain constant data, like tables, etc., you should also consider choosing another file number for these segments. Then Adabas FASTPATH can work in a more efficient way.

In order to access dependent segments with Natural applications, you may include descriptors and superdescriptors in the Adabas file. You can do that by adding an entry to the ADACMP cards which are produced by the ADL CBC utility. Once the data has been converted into the ADL file, you can use the Adabas ADAINV utility in order to generate descriptors, superdescriptors etc. Such a superdescriptor might consist of the PCKs of all parent segments and the sequence field of the segment to be accessed. If this segment does not have a sequence field, the super-descriptor would also return data of other segment types under the same parent. In order to avoid this, you can choose another file number during the CBC utility run for the affected segment.

4 ADL Conversion Utilities for DBDs and PSBs

For the Adabas Bridge for DL/I to be able to process a DL/I call, it must be aware of the original DL/I structures and the rules according to which they were defined, and how the data is structured under Adabas. This information is in ADL control blocks stored in the ADL directory file. The control blocks are created during the DBD/PSB conversion process, which uses the original DL/I DBD and PSB sources as input. These are assembled and then processed further by the CBC utility.

For an application program to run against ADL, both the PSB used and the DBDs which it references must be run through the DBD/PSB conversion process. This section describes the procedure for generating the necessary ADL control blocks from the DBD/PSB macro source.

This chapter covers the following topics:

Conversion - Overview

To convert a DL/I DBD or PSB into an ADL DBD or PSB, you must perform the following steps:

All PSBs and DBDs except Primary Index DBDs

Step	Description
Step 1	Assemble and link edit the original DL/I DBD or PSB source.

Step 1

Assemble and link edit the original source of the DL/I DBD or PSB using the ADL macros provided in the Source Library on the installation tape. You may use the sample JCL contained in the members `ADLDPC1` (z/OS) or `ADLDPC1.J` (z/VSE) for this. For the special link-edit requirements of an HD database, see the corresponding topic later in this section.

In addition to the DL/I keywords you can add the “`PRINT`” keyword to the DBD macro or to the first PCB macro of a PSB.

PRINT=GEN	generates the full assembler listing,
PRINT=NOGEN	is the default and generates the short output.

All PSBs and Physical and Logical DBDs

Step	Description
Step 2	Run the CBC utility once for each DBD/PSB.

Step 2

Run the CBC utility once for each DBD and PSB processed in Step 1. For z/OS, you may use the sample JCL contained in the members `ADLDPC23` for physical DBDs (contains Step 3 as well) or `ADLDPC2` for PSBs and logical DBDs. z/VSE users should consult members `ADLDPC23.J` or `ADLDPC2.J` respectively.

Sample JCL/JCS requirements for the CBC utility can be found at the end of this section.

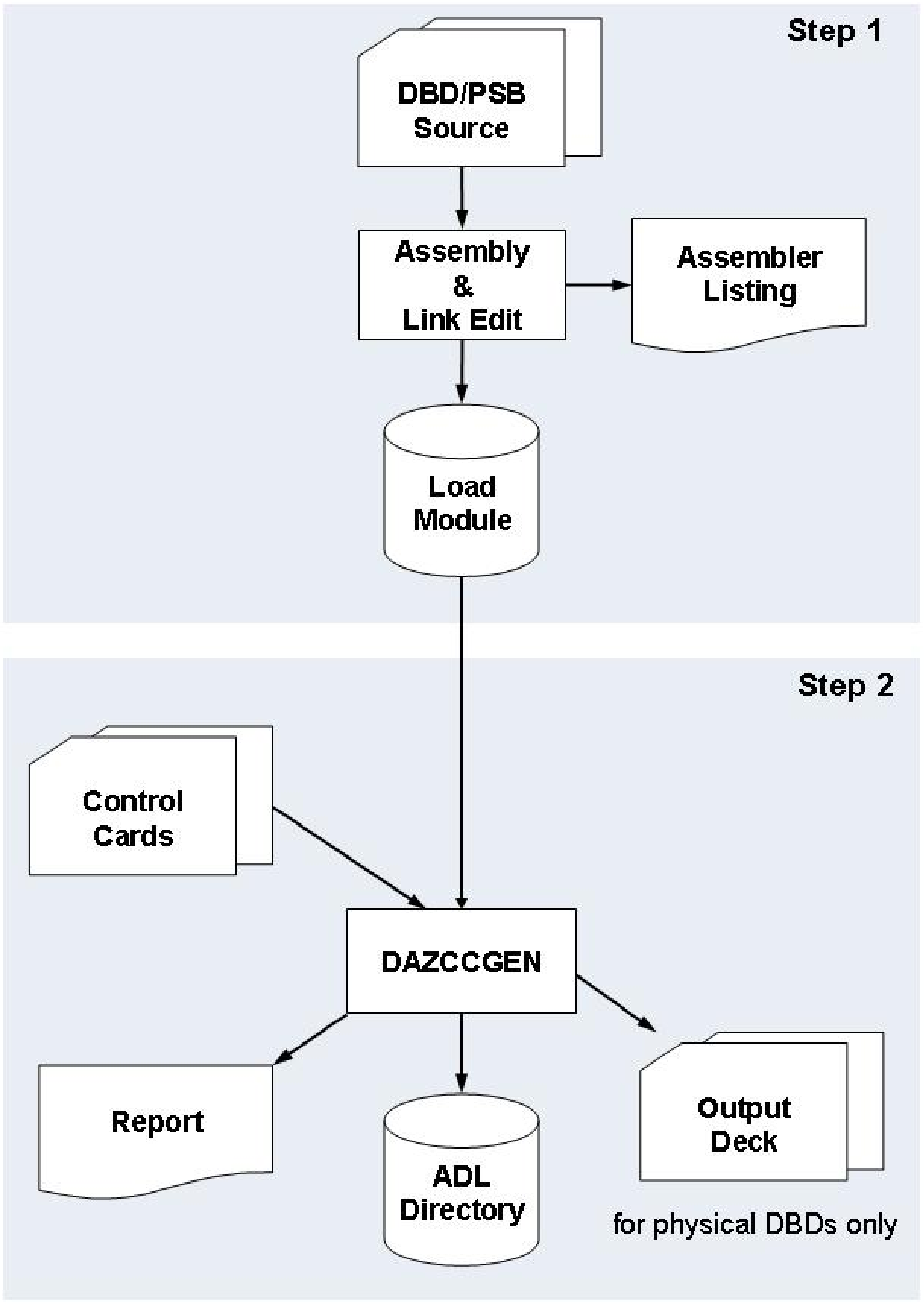
The following input must be provided for each DBD or PSB:

- The load module produced in Step 1;
- One or more control statements, as described later in this section.

The following output is produced:

- An ADL DBD or PSB entry in the ADL directory file;
- The `ADACMP` statements for the Adabas file(s) to be used to store the converted data (for physical DBDs only);
- The Adabas User Exit 6 extensions to be used during initial loading of the Adabas file(s) (for physical DBDs only);
- A report describing the original DL/I structure, the new Adabas structure, and the relationship between the two (for DBDs only).

DBD/PSB Conversion Steps 1 - 2


Step 1
DBD/PSB Source
Assembly & Link Edit
Assembler Listing
Load Module
Step 2
Control Cards
DAZCCGEN
Report
ADL Directory
Output Deck
for physical DBDs only

Physical DBDs Only

Step	Description
Step 3	Create input decks.
Step 4	Assemble and link-edit the Adabas User Exit 6 extension.

Step 3

The output deck created by the CBC utility contains several members (ADACMP control statements and the User Exit 6 extensions for each Adabas file). If the CBC utility has generated output control statements (see the description of the `UTI` parameter in *ADL Parameter Module* in the *ADL Installation* documentation), you can use one of the IBM utilities `IEBUPDTE` (z/OS) or `LIBR` (z/VSE) to create the members. The control statement output data sets generated by the CBC utility are named according to the following conventions:

```
xfffff
```

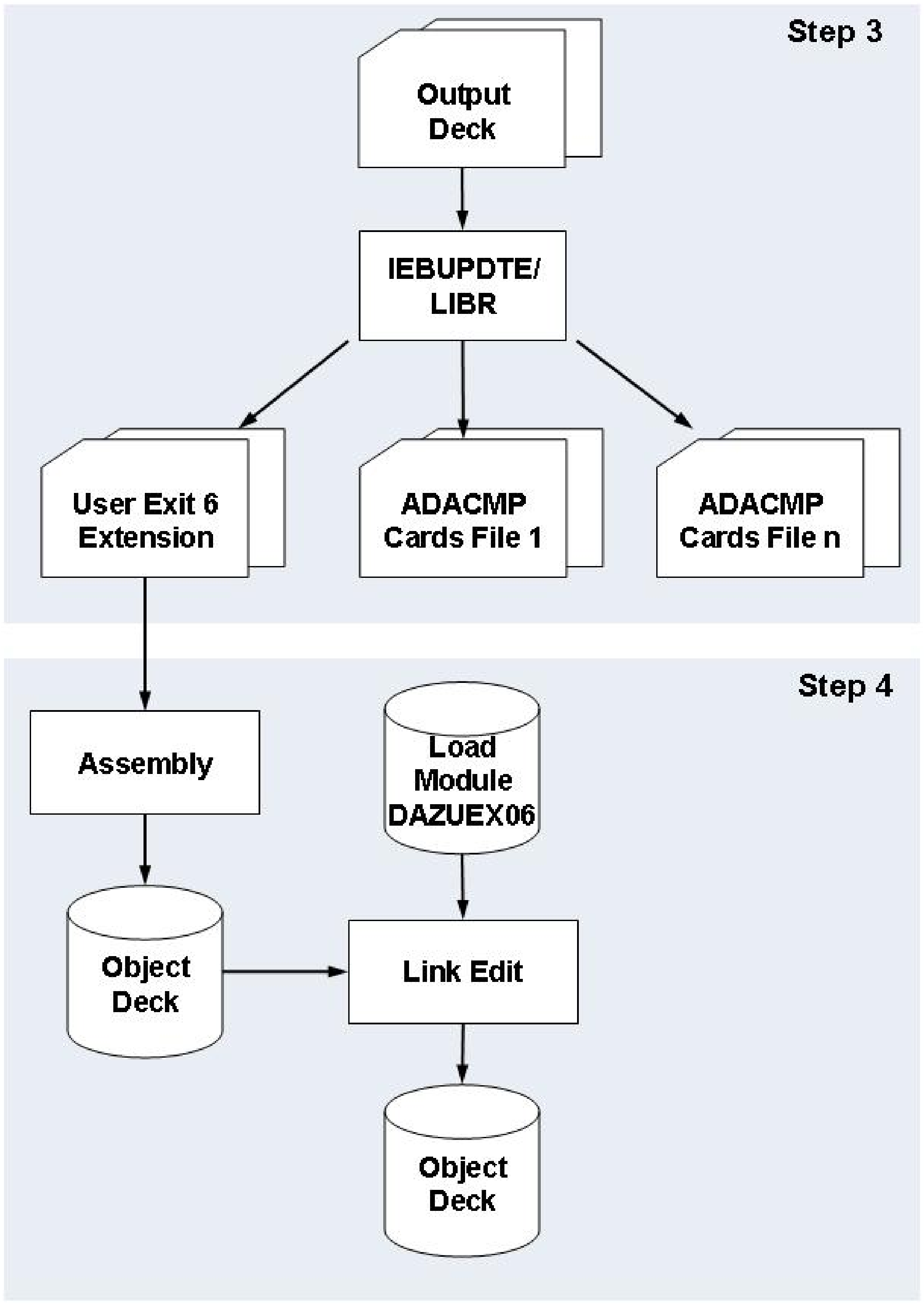
The individual identifiers are as follows:

Identifier	Explanation
x	W for ADACMP control statements I for the Adabas User Exit 6 Extension
fffff	The five-digit number of the Adabas file used to store the root segment data.

Step 4

Assemble the Adabas User Exit 6 Extension (i.e. the output from Step 3) and link edit this with the fixed part, DAZUEX06. You may use the sample JCL in the Source Library member ADLDPC4 (z/OS) or ADLDPC4.J (z/VSE) as an example.

DBD/PSB Conversion Steps 3 - 4


Step 3
Output Deck
IEBUPDTE/ LIBR
User Exit 6 Extension
ADACMP Cards File 1
ADACMP Cards File n
Step 4
Assembly
Load Module DAZUEX06
Object Deck
Link Edit
Object Deck

The control blocks to be converted are of different types, such as PSBs, physical DBDs and logical DBDs. Not all the steps in the DBD/PSB conversion process need to be performed for all control block types. The table below provides an overview of which steps are required for which control blocks and is followed by more detailed explanations.

Type of PSB/DBD	Step 1	Step 2	Step 3	Step 4
Physical DBD	YES	YES *	YES *	YES
Primary Index DBD	NO	NO	NO	NO
Secondary Index DBD	YES	NO	NO	NO
Logical DBD	YES	YES	NO	NO
PSB	YES	YES	NO	NO

* Note that the sample JCL(JCS) in member ADLDPC23 (ADLDPC23.J) enables both steps to be run in a single job.

It is recommended to convert all PSB and logical DBD control blocks once, whether the physical DBDs referenced by them are already converted or not. ADL will automatically detect when a physical DBD is not yet converted and will direct the corresponding database call to DL/I.

This prevents you from monitoring which PSB in detail references which DBD. The only thing you have to do when you convert a further database is to convert the corresponding DBD control block. In a CICS environment, you will have to repeat some of the steps described in the topic *Generating the Runtime Control Tables* in the *ADL Interfaces* documentation.

Conversion - PSBs and all DBDs except Index DBDs

Assemble and link edit all DBDs, except for primary index DBDs, and all PSBs using the ADL macros provided in the ADL Source Library on the installation tape. You may use the sample JCL(JCS) contained in the members ADLDPC1 (ADLDPC1.J) for this. Primary index DBDs and PSBs which do not contain DB-PCBs are not used by ADL and may, therefore, be omitted from the DBD/PSB conversion process. After this, run all DBDs and PSBs except secondary index DBDs through the CBC utility.

If you do not intend to convert all DL/I databases at once, you need to convert only those DBDs describing the databases being converted. Likewise, you need to convert only those PSBs which reference converted data.

PSBs that reference both converted and non-converted databases are used in the mixed mode environment.

Conversion - Physical DBDs

If a physical DBD is to be run through the CBC utility more than once, you must delete it from the ADL directory file first. This must be done to ensure that the latest version of the DBD is taken (i.e. the version in the Load Library and not the one stored in the ADL directory file).

You may delete control blocks for a physical DBD stored in the ADL directory file using the DELDBD function described later in this section, or with the ADL Online Services.

For physical DBDs, Steps 3 and 4 must also be performed in order to complete the conversion process.

Conversion - Index DBDs

All index DBDs, except for the primary index DBD for `HIDAM` or `HISAM` databases, need to be run through the assembly and link edit process described in Step 1. No further steps need to be performed.

The index DBD control blocks are automatically processed and stored in the ADL directory file when the physical DBD for which the secondary index has been defined is run through the CBC utility. This means that you must run all secondary index DBDs through the assembly and link edit process described in Step 1 before you can successfully run a DBD containing secondary indices through the CBC utility. Each secondary index definition will result in the generation of an Adabas descriptor as part of the Adabas file description for the file containing the source segment data.

If an index DBD contains user data fields, Adabas fields will automatically be generated as part of the Adabas file description of the file containing the source segment data.

Once stored in the ADL directory file, the control blocks for an index DBD may be deleted like any other DBD. If you delete a physical DBD with the ADL Online Services, you should also delete all related index DBDs in order to avoid problems which could arise during regeneration.

Where a physical DBD containing secondary indices is to be run through the CBC utility more than once, the ADL control blocks for the secondary index DBDs will be deleted and re-created automatically.

Conversion - Logical DBDs

You must run all logical DBDs through the assembly and link edit process as described in Step 1 and then through the CBC utility as described in Step 2. The only parameter required for the CBC utility is the name of the logical DBD to be converted. All other parameters will be ignored.

No `ADACMP` statements or Adabas User Exit 6 extensions are generated for logical DBDs. This is because logical DBDs are based on database(s) which are described by physical DBDs, and these have to be run through the complete conversion process separately. All `ADACMP` statements and Adabas User Exit 6 extensions needed are generated then.

Once stored in the ADL directory file, the control blocks for a logical DBD may be deleted in the same way as any other DBD.

Conversion - Logically Related Physical DBDs

Logically related physical DBDs reference each other using definitions of logical child and logical parent segment types. When processing a physical DBD, the CBC utility needs access to all other physical DBDs which are logically related to the DBD being processed. The CBC utility first tries to locate a related DBD in the ADL directory file. If this fails, the CBC utility tries to load the related DBD as a load module from the Load Library. If this also fails, the CBC utility reports an error.

This means that before you can successfully run a DBD containing logical relationships through the CBC utility, you have to run all other related physical DBDs through the assembly and link edit process described in Step 1 at the beginning of this section.

Running a physical DBD containing logical relationships through the CBC utility stores both the physical DBD being processed and all other logically related physical DBDs in the ADL directory file. However, this does not complete processing of these ADL control blocks; you still have to run each one of them separately through the CBC utility. You can see whether processing has been completed for all DBDs referenced by the DBD being run through the CBC utility by consulting the list provided at the end of the CBC utility report.

If a physical DBD is logical related to other DBDs, you must specify the `LOGID` parameter at the `GENDBD` function for a unique identification of the DBD's data.

If a physical DBD is to be run through the CBC utility more than once, you must first delete it and all other logically related physical DBDs from the ADL directory file. Then re-convert them all again (without further intermediate deletion). This ensures that the latest version of the DBDs is used (that is, the version in the Load Library and not that stored in the ADL directory file).

Conversion - HD Databases

For HD databases an alternate sequence is defined by the ACCESS macro, which replaces the secondary index DBDs. When such a DBD source is assembled, the result is one module containing the physical DBD as well as the secondary index DBDs. Since ADL expects the secondary index DBDs on separate members, this module must be split up, as described below. The remaining steps 2 - 4 of the conversion can be performed in the common way, as described earlier in this section.

z/OS Requirements

Add the following entries to the link-edit for the physical DBD:

```
ENTRY   dbdname
REPLACE     secname-1
.
.                   for each secondary index
.
REPLACE     secname-n
INCLUDE     OBJMOD
NAME    dbdname(R)
```

for each secondary index add the following entry:

```
ENTRY   secname-x
REPLACE dbdname
REPLACE secname-1
.
.                   all secondary indices, but not secname-x
.
REPLACE     secname-n
INCLUDE     OBJMOD
NAME        secname-x(R)
```

z/VSE Requirements

Assemble the DBD with option “DECK” to create an OBJ module and then add the following entries to the link-edit for a physical DBD:

```
PHASE dbdname,*,NOAUTO
INCLUDE dbdname,(dbdname)
```

and for each secondary index add the following entry:

```
PHASE secname-x,*,NOAUTO
INCLUDE dbdname,(secname-x)
```

Control Statements for the CBC Utility

As mentioned earlier, you must provide a control statement for each DBD and PSB processed using the CBC utility. Each control statement must have the following format:

```
function p1,p2... comments
```

The individual parameters are as follows:

Parameter	Explanation
function	A function keyword. It must start in the first column of a statement.
p1,p2...	Parameters for the function given. They must follow the function and be separated from it by at least one blank. No blanks should be left between parameters.
comments	Comment statements. Comments can be made on statements by leaving at least one blank after the last parameter, or by inserting an asterisk (“*”) in the first column of a statement.

The various function keywords are as follows:

Function	Parameters
GENDBD	NAME=DBD-name, LOGID=logical-Id, DBID=dbid, FNR=file-number, TYPE=conversion-type, CONSI=YES/NO
GENSEG	NAME=segment-name, LOGID=logical-Id, FNR=file-number, ADANAME=Adabas-short-name, BACKW=YES/NO
DELDBD	NAME=DBD-name
GENPSB	NAME=PSB-name
DELPSB	NAME=PSB-name

These functions are described in more detail in the following topics:

- GENDBD Function
- GENSEG Function
- DELDBD Function

- GENPSB Function
- DELPSB Function

GENDBD Function

The `GENDBD` function initiates the processing of the DBD to be converted. Its parameters are as follows:

Parameter	Description
NAME	The name of the ADL DBD to be converted. It can be between 1 and 8 alphanumeric characters long; the first character must be a letter. The name specified must be the same as the original DL/I name of the DBD.
LOGID	(for physical DBDs only) The default logical ID of the DBD. If the DBD is involved in logical relationships, the `LOGID` given must be different from the `LOGID`s of the other DBDs. Possible values: 1 - 255 Default: 1
DBID	(for physical DBDs only) The Adabas database ID for the database which will be used to store the converted data. If it is omitted, the `DBID` of the ADL directory will be used. If the `DBID` is the same as the `DBID` of the ADL directory, it is recommended to omit the `DBID` parameter. This eases the creation of mirror databases. Possible values: 1 - 65535 Default: none
FNR	(for physical DBDs only) The default Adabas file number for the file which will be used to store the converted data for all segment types not affected by a `GENSEG` function. Possible values: 1 - 65534 Default: none
TYPE	(for physical DBDs only) This parameter is used to indicate whether the database being processed is to be converted to Adabas or not. Possible values: `ADA` the database will be converted `DLI` the database will reside under DL/I Default: `ADA`
CONSI	(for physical DBDs only) This parameter specifies whether the converted DBD may be accessed by Natural/Adabas applications via the ADL Consistency Interface or not. The assignment can be modified later using the ADL Online Services. The `CONSI` parameter must be set to `YES` if the DBD is accessed by both DL/I and Natural/Adabas applications. For more information, refer to the section Consistency DBD Maintenance in the section *ADL Online Services* in the *ADL Interfaces* documentation . Possible values: YES = DBD used by ADL Consistency Interface NO = DBD not used by ADL Consistency Interface. Default: `YES`

Note: If the DBD was originally converted with ADL version 2.2 or before, the LOGID should specify the same value as the ADL 2.2 FNR parameter.

The SEQ (processing sequence) parameter of ADL 2.2 has become obsolete. The new layout of the ADL internal pointer field has the same performance advantages as the previous "SEQ=SEG" setting.
To avoid conflicts with the logical files settings (LFILE) of Natural, it is recommended not to use DBID=255.

GENSEG Function

The `GENSEG` function is used to control DBD conversion for a DL/I segment. It may be specified for each segment of a physical DBD but not for virtual logical child segments. If no `GENSEG` function is specified for a particular segment, then the default parameter values are used.

The function parameters are explained below.

Parameter	Description
NAME	The name of the segment in the DBD currently being processed. This may be between 1 and 8 alphanumeric characters long. The first character must be a letter.
LOGID	The logical ID of the segment. The number given will be used as default for all dependent segment types. If the DBD is involved in logical relationships, the `LOGID` given must be different from the `LOGIDs` of the other DBDs. Possible values: 1 - 255 Default: `LOGID` of parent segment or `LOGID` of `GENDBD` function for the root segement
FNR	The Adabas file number for the file which will be used to store the converted data of this segment type. The number given will be used as the default for all dependent segment types. Possible values: 1 - 65534 Default: none
ADANAME	The Adabas short name to be used as the name of the Adabas field group which will store the segment data. If the `ADANAME` parameter is not specified, then ADL generates the Adabas group names automatically.
BACKW	Used to specify whether the Adabas descriptor used for internal reading backwards is to be maintained for the segment being generated. This is done in order to optimize retrieval of the last segment occurrence in a twin chain. Possible values: YES = indicates that the descriptor will be maintained; NO = indicates that it will not. The descriptor is maintained automatically where the physical insert rule is either `HERE` or `LAST`.

Note: If the DBD was originally converted with ADL version 2.2 or before, the LOGID should specify the same value as the original FNR parameter. Otherwise it is not required to specify LOGIDs for segments even if the DBD data is saved on multiple Adabas files.

DELDBD Function

The `DELDBD` function deletes an existing DBD from the ADL directory file.

The function parameter is as follows:

Parameter	Description
NAME	The DL/I name of the DBD to be deleted. This may be 1 to 8 alphanumeric characters long. The first character must be a letter.

GENPSB Function

The `GENPSB` function initiates processing of the PSB specified.

The function parameter is as follows:

Parameter	Description
NAME	The DL/I name of the PSB to be converted. This may be 1 to 8 alphanumeric characters long. The first character must be a letter.

DELPSB Function

The `DELPSB` function deletes an existing PSB from the ADL directory file.

The function parameter is as follows:

Parameter	Description
NAME	The DL/I name of the PSB to be deleted. This may be 1 to 8 alphanumeric characters long. The first character must be a letter.

CBC Utility Output

The CBC utility produces up to four different types of output:

1. *ADL control blocks on the ADL directory file.*

   The CBC utility stores and updates the ADL DBDs and PSBs on the ADL directory file.
2. *Output deck containing ADACMP statements and the Adabas User Exit 6.*

   This output is only produced when the CBC utility is processing a physical DBD. The ADACMP statements for all files used to store the data and the Adabas User Exit 6 extension for loading the data into the Adabas file(s) are produced as one output deck. Each set of records in the deck is separated from the others by control statements. These can be interpreted by an IBM utility (LIBR for z/VSE and IEBUPDTE for z/OS) to create separate members in a library.

Note that the generation of output control statements by the CBC utility can be suppressed by specifying the parameter UTI=(,N). See the chapter The ADL Parameter Module in the ADL Installation Manual for further details.

3. *Control print output.*

   A print file giving the input control statements for the CBC utility and the action taken. It has the following format:

   nnnnn FUN input statement ACT message

   where *nnnnn* is the number of the input statement.

   The input statements are followed by information on the run, including the number of report pages produced and the number of input statements.

4. *Report (physical and logical DBDs only).*

   In the case of physical DBDs, the report contains a complete overview of the DL/I and Adabas structures involved. A list of all DBDs referenced by the DBD just processed, i.e. secondary index DBDs and other physical DBDs related via logical relationships, is given at the end of the report. The list also indicates whether processing of the DBDs referenced has been completed or not.

z/OS Requirements

The following table lists the data sets used by the CBC utility `DAZCCGEN`.

DDname	Medium	Description
DAZIN1	Reader	Control input for the ADL batch monitor, DAZIFP.
DAZOUT1	Printer	Messages and codes.
DAZOUT2	Printer	Report.
DAZOUT4	Disk	ADACMP statements and Adabas User Exit 6 extension.

Example:

The following is an example of a job to run the CBC utility:

```
//          EXEC PGM=DAZIFP,PARM='UTC,DAZCCGEN'
//STEPLIB   DD DSN=ADLxxx.LOAD,DISP=SHR
//          DD DSN=ADABAS.Vnnn.LOAD,DISP=SHR
//DDCARD    DD *
ADARUN PROGRAM=USER,...
//DAZIN1    DD *
DELDBD NAME=COURSEDB
GENDBD NAME=COURSEDB,DBID=009,FNR=034
//DAZOUT1   DD SYSOUT=X
//DAZOUT2   DD SYSOUT=X
//DAZOUT4   DD DSN=&&DECK,DISP=(,PASS),UNIT=SYSDA,
//             SPACE=(80,(100,100),RLSE),
//             DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//*
//          EXEC PGM=IEBUPDTE,PARM=NEW
//SYSPRINT  DD SYSOUT=X
//SYSUT2    DD DSN=ADLxxx.SOURCE,DISP=SHR

//SYSIN     DD DSN=&&DECK,DISP=(OLD,DELETE)
```

z/VSE Requirements

The following table lists the files used by the CBC utility, `DAZCCGEN`.

DTF	Logical Unit	Medium	Description
DAZIN1	SYSIPT	Reader	Control input for the ADL batch monitor, DAZIFP.
DAZOUT1	SYSLST	Printer	Report, messages and codes.
DAZOUT2	SYS011	Printer	Report. *
DAZOT3D	SYS013	Disk	Report. **
DAZIN3D	SYS014	Disk	Report. **
DAZOUT4	SYSxxx	Disk	ADACMP statements and Adabas User Exit 6 extension.

* Only required when more than one logical printer is available. In this case, `SYS011` may be used to assign a second printer to which the report will be routed directly.

** Only required when only one logical printer is available. In this case, the report which is normally directed to `DAZOUT2` as the second print file will be written to disk. At the end of the job it will be read from disk and routed to `DAZOUT1`.

The control input for the batch monitor (`DAZIFP`), for `ADARUN`, and for the CBC utility itself are all read from `SYSIPT`. The control statements for this must be specified in the following order:

```
UTC,DAZCCGEN,,...                        input for DAZIFP
/*
ADARUN DB=dbid,MO=MULTI,PROGRAM=USER,... input for ADARUN
/*
DELDBD NAME=dbd                          input for the CBC utility
GENDBD ...
.
/*
```

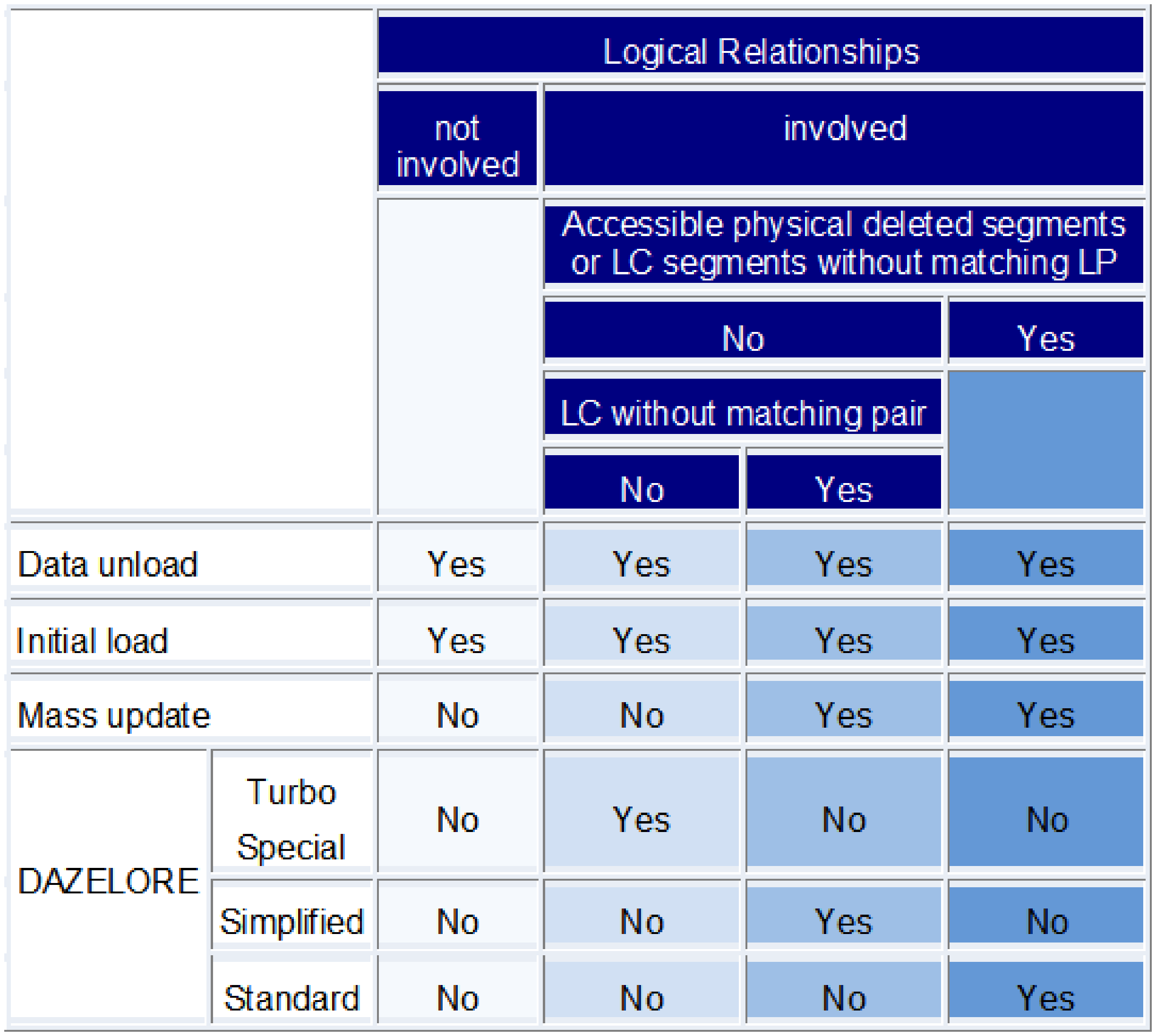
Example:

The following is an example of a job to execute the CBC utility:

```
// ASSGN SYS010,DISK,VOL=volser,SHR
// DLBL DAZOUT4,'punchfile',0,SD
// EXTENT SYS010,volser,......
// ASSGN SYS013,DISK,VOL=volser,SHR
// DLBL DAZOT3D,'printfile',0,SD
// EXTENT SYS013,volser,......
// DLBL DAZIN3D,'printfile',0,SD
// EXTENT SYS014,volser,......
// EXEC DAZIFP
UTC,DAZCCGEN
/*
ADARUN PROGRAM=USER,..........
/*
DELDBD  NAME=COURSEDB
GENDBD  NAME=COURSEDB,DBID=9,FNR=34
/*
// DLBL IJSYSIN,'punchfile'
// EXTENT SYSIPT,volser
ASSGN SYSIPT,DISK,VOL=volser,SHR
// EXEC LIBR,PARM='ACCESS S=SAGLIB.ADL...'
/&
CLOSE SYSIPT,FEC
```

5 Conversion of the Data - General Considerations

Once the DL/I hierarchical data structures are converted into an Adabas file layout, the data may be moved to Adabas. The steps which are necessary to convert the data depend on the DBDs and user applications involved. The following table lists the necessary steps according to the different prerequisites:



		Logical Relationships			
		not involved	involved		
			Accessible physical deleted segments or LC segments without matching LP		
			No		Yes
			LC without matching pair		
			No	Yes	
Data unload		Yes	Yes	Yes	Yes
Initial load		Yes	Yes	Yes	Yes
Mass update		No	No	Yes	Yes
DAZELORE	Turbo Special	No	Yes	No	No
	Simplified	No	No	Yes	No
	Standard	No	No	No	Yes

This chapter covers the following topics:

Conversion of the Physical Hierarchical Structures

The data conversion of a hierarchical structure with no logical relationship involved is quite simple. The data has to be unloaded with the ADL unload utilities and then loaded into an Adabas file, using standard Adabas utilities.

Two different operations are required in order to unload the data from the DL/I database. The first is to read the data and the second is to prepare it for the loading into Adabas.

ADL offers two different methods for performing these steps: an automated and a manual procedure. For the automated procedure, both steps are combined in the utility `DAZUNDLI`, while the manual procedure splits them up into two utilities, namely `DAZUNLOD` and `DAZREFOR`.

It is recommended to use the automated procedure as far as possible. There is no need to customize, assemble or link-edit the program, and no overhead caused by writing and reading the intermediate unload file as with the manual procedure.

Nevertheless, it can be advantageous to choose the manual procedure in some cases. One reason could be that the limited data editing capability of the automated procedure is not sufficient for your applications. Another reason could be that the DL/I system and the ADL nucleus (or Adabas) are not available at the same time. In this case you cannot use the automated procedure, because `DAZUNDLI` accesses both DL/I and ADL. On the DL/I side you have to use `DAZUNLOD`, which runs as a normal DL/I application program. Then move the intermediate unload file to the ADL / Adabas side and run it against `DAZREFOR`, which is a normal mode ADL application program.

Regardless of which method was used for unloading the data, it will be initially loaded into an Adabas file by using the Adabas utilities `ADACMP` and `ADALOD` (`LOAD` function). If no logical relationship is involved, then the data conversion process is finished with the `ADALOD` run.

Additional Effort Related to Logical Relationships

If the database is involved in logical relationships, up to two further steps are required in addition to the unloading and the initial loading of the data. The first is the mass update for paired logical child segments and the second is the establishing of logical relationships.

The mass update is only required if the database is involved in bi-directional logical relationships and if the simplified or standard procedure is used to establish logical relationships. You may use the standard Adabas utilities `ADACMP` and `ADALOD` (`UPDATE` function) in order to perform the mass update.

To establish logical relationships, ADL offers four different procedures: Standard, Simplified, Special and Turbo. All of these procedures use the ADL utility `DAZELORE`. The differences between them are described in detail in the next section.

The Standard procedure can be used in any case, but it is the most time consuming procedure. On the other end, the Turbo procedure has the highest performance but also the most restrictions.

Thus you should use the Turbo procedure whenever possible, or when the Turbo procedure cannot be used, the Special procedure, then the Simplified. Only in cases which cannot be satisfied by any of these three procedures, use the Standard procedure.

Validating Data Types

As described in the section *Conversion of the Data Structures - General Considerations* in this documentation, a DL/I field can contain non-numeric data, even if it is defined as numeric. ADL will convert the DL/I field definition to a numeric Adabas field. During the loading of the data, Adabas will reject the non- numeric values, and thus the load fails.

The standardized ADL unload utility `DAZUNDLI` offers you a limited editing capability for such incorrect data. If you specify the `MODE=CHECKNUM` or the `SEGM/FIELD` parameters, all numeric fields or the specified numeric fields will be checked for valid contents. If ADL detects an incorrect value, a null value (zoned decimal or packed zero depending on the field type) is substituted.

If this limited data editing is not sufficient for you, the customized ADL unload utility `DAZUNLOD` may be used after modifying it as desired.

Nevertheless, to avoid all such failures during the load, you should consider adapting the DL/I field definitions prior to the conversion, as described in the section mentioned above.

6 ADL Data Conversion Utilities

This chapter covers the following topics:

Overview

There are two ways in which the data stored in a DL/I database can be converted to Adabas file(s): an automated procedure and a manual one. Both procedures create a sequential file (the unloaded database), which is input to the standard Adabas Compression utility, `ADACMP`. After the data compression by `ADACMP`, the Adabas utility `ADALOD` (Initial File Loading and Mass Update Utility) will populate the Adabas file(s) using the functions `"LOAD"` and `"UPDATE"`.

The difference between the two procedures lies in the fact that the automated procedure uses a standardized Unload utility, `DAZUNDLI`. This utility unloads the original DL/I database in one step. In contrast, the manual procedure creates the unloaded database in two steps, using a customized Unload utility, `DAZUNLOD`, in the first step, and a standardized Reformat utility, `DAZREFOR`, in the second.

In the automated procedure, the Unload utility `DAZUNDLI` accesses DL/I to read the segments in the database, while at the same time using the ADL to create the unloaded database. The utility thus runs in mixed mode, and the PSB and DBD used need to be generated according to mixed mode conventions (i.e. as both DL/I and ADL PSBs and DBDs). Limited data editing is possible during unloading of the DL/I database: all or specific numeric fields may be checked for valid numeric contents. You can limit the amount of unloaded data by specifying various parameters.

The manual procedure uses a customized Unload utility, `DAZUNDLI`, to unload the original DL/I database, and a standardized Reformat utility, `DAZREFOR`, to create the unloaded database. `DAZUNLOD` is a normal DL/I application program which reads the DL/I database from the beginning to the end and creates an intermediate unload file. `DAZREFOR` is a normal mode ADL application program which reads the intermediate unload file and creates the input file for `ADACMP`. The PSB and DBD used for both `DAZUNLOD` and `DAZREFOR` may be the same.

As you have to customize `DAZUNLOD` in order to unload a DL/I database, you are also able to edit the data read before you create the intermediate unload file. You may, for example, want to do this in situations where numeric data has not been stored in numeric fields.

The differences between the automated and the manual procedure are summarized on the following pages.

All PSBs created for the ADL utilities mentioned in this section (for example, the unload and connection PSBs) must be installed with the option `LANG=ASSEM` or `LANG=COBOL` in the `PSBGEN` statement.

Automated Procedure	Manual Procedure
The ADACMP input file is created in one step.	The ADACMP input file is created in two steps.
The Unload utility runs in mixed mode	The Unload utility runs as a normal DL/I application. The Reformat utility runs as a normal ADL application
Limited data editing possible during the unload procedure.	The Unload utility has to be customized. Data editing is therefore possible.

Data Unload with the ADL Unload Utility

This section describes how to unload all the data of a database. Refer to the next section for details on how to unload data selectively.

To use the automated method of conversion to unload the data stored in a DL/I database, perform the following steps:

- Step 1: Convert the Physical DBD
- Step 2: Create an ADL Unload PSB
- Step 3: Create a DL/I Unload PSB
- Step 4: Unload the Data

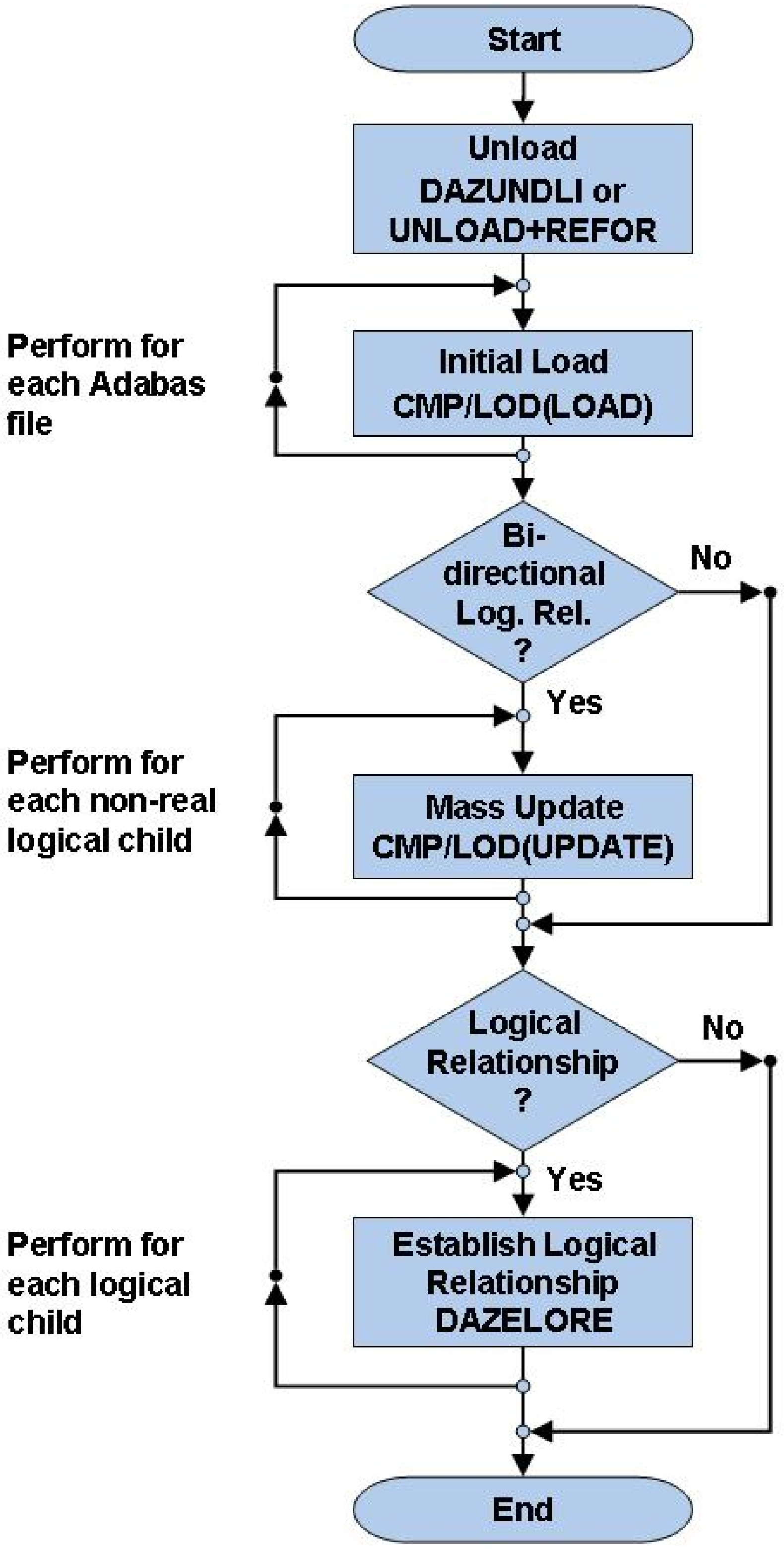
Step 1: Convert the Physical DBD

Run the original DL/I DBD through the CBC utility (see the section *ADL Conversion Utility for DBDs and PSBs* in this documentation).

Step 2: Create an ADL Unload PSB

The PSB created must contain two PCBs, both of which must be based on the original DBD and must reference all of its segments. The first PCB is used to unload the data, while with the help of the second, the data is prepared for the reloading. Run this PSB through the CBC utility. Note that if the DBD is involved in logical relationships and contains logical child segments, these must also be referenced by the PCBs (this also applies to virtual logical child segments).

Data Base Conversion Overview


Start
Unload
DAZUNDLI or
UNLOAD+REFOR
Perform for each Adabas file
Initial Load
CMP/LOD(LOAD)
Bi-directional Log. Rel. ?
No
Yes
Perform for each non-real logical child
Mass Update
CMP/LOD(UPDATE)
Logical Relationship ?
No
Yes
Perform for each logical child
Establish Logical Relationship
DAZELORE
End

Step 3: Create a DL/I Unload PSB

Run the PSB through the DL/I `PSBGEN`. The PSB created in the previous step can be used as the DL/I unload PSB. Note that, in contrast to the previous step, it is not absolutely necessary for this PSB to contain two identical PCBs, as one suffices.

Step 4: Unload the Data

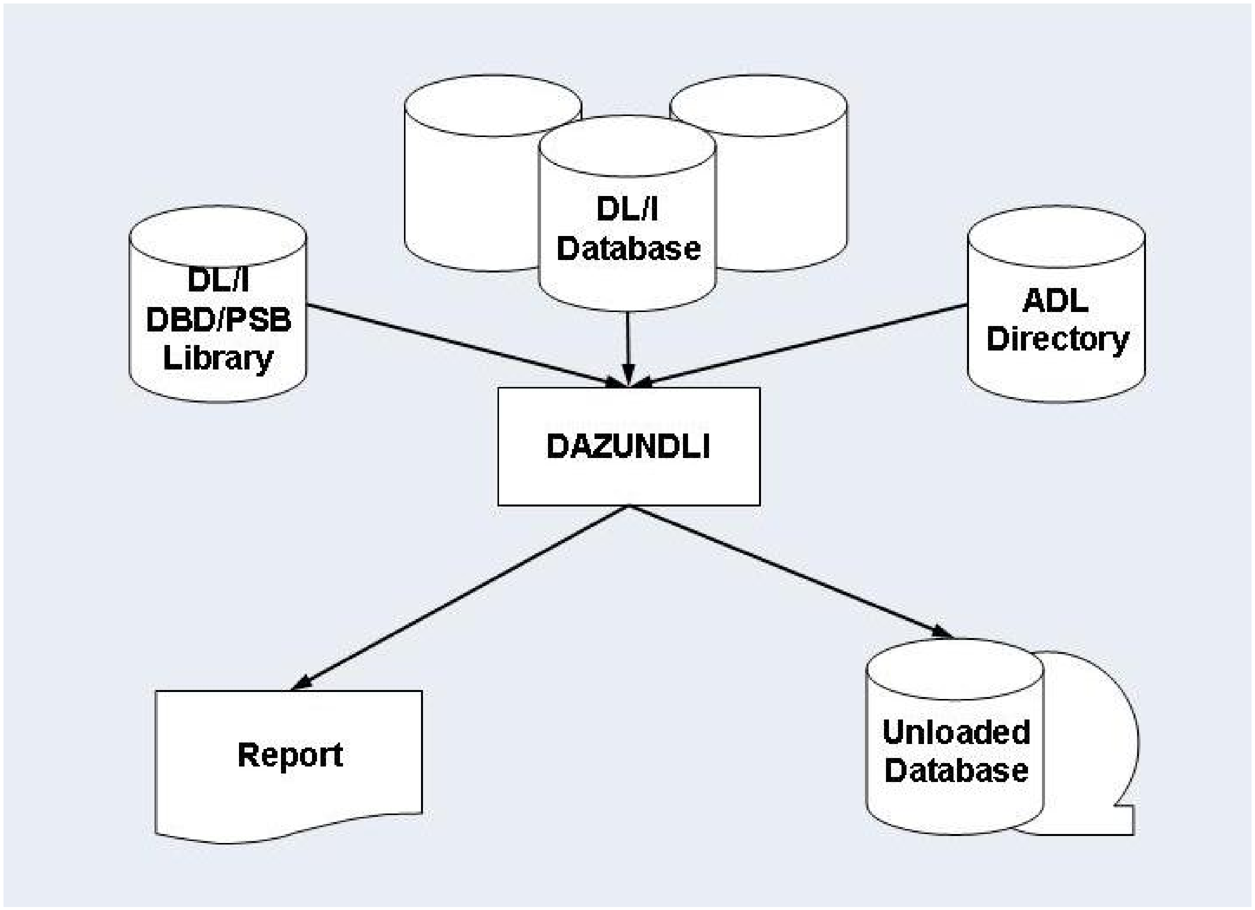
Unload the data from the DL/I database by running the ADL Unload utility, `DAZUNDLI`. `DAZUNDLI` is executed as a mixed mode program (see the section *Batch Installation and Operation* in the *ADL Interfaces* documentation). The mixed mode control statement must have the following layout:

```
UNL,DAZUNDLI,psbname
```

where *psbname* is the name of the unload PSB.

The unloaded data is subsequently stored in a sequential file. You can use the sample JCL in the source library member `ADLDBC4` (z/OS) or `ADLDBC4.J` (z/VSE) as an example. The JCL/JCS requirements for `DAZUNDLI` are given at the end of this section.

Database Conversion Unload (Automated Procedure)



Data Validation

During unloading of the DL/I database contents, it is possible to let ADL automatically correct either all numeric fields or specific numeric fields. Packed (`TYPE=P`) and zoned decimal (`TYPE=Z`) fields are checked by ADL for valid packed or zoned decimal value contents. If they do not contain a valid packed or zoned decimal value, a null value (i.e. a packed zero for packed fields and a zoned decimal zero for zoned decimal fields) is substituted. This procedure can be activated for all numeric fields within a DBD or for specific ones only, using extra control cards for `DAZUNDLI`. The syntax of these control cards is as follows:

```
MODE=CHECKNUM
SEGM=segname,FIELD=fldname
SEGM=
```

where *segname* is the name of segment within the unloaded DBD, and *fldname* is the name of a numeric field within this segment.

Specifying only the control card `MODE=CHECKNUM` activates checking of all numeric fields within the DBD. Specifying one or more `SEGM/FIELD` control cards limits the checking to only those fields specified. Omitting the control cards altogether, or specifying `MODE=STANDARD`, deactivates checking of any numeric fields.

Limited Data Unload

The automated procedure described in the previous section may also be used to unload only a part of the database. This can be achieved by modifying the unload PSB or by specifying additional control statements for the `DAZUNDLI` utility. Note that if a logical child segment occurrence is unloaded, the corresponding logical parent segment occurrences have to be unloaded as well.

This section covers the following topics:

Restrict the Unload Segment Types

If the first PCB in the unload PSB does not contain all of the segments of the original DBD, only the referenced ones will be unloaded. The second PCB has to reference at least the segments of the first PCB. Note that a logical relationship between two DBDs requires a specification of the logical child segment either on both sides or on none.

Specify an Alternate Unload Sequence

In order to unload the database in an alternate sequence, the first PCB in the unload PSB may refer to a secondary index as the processing sequence. In this case the "`START`" and "`END`" parameters of `DAZUNDLI` correspond to the values of the secondary index source fields.

The secondary index must have the root segment as target. It is recommended to use only those secondary indices which have a one-to-one relation to the root segment, otherwise the repetition of the data will lead to problems during the reload.

Limit the Unloaded Records

The number of unloaded records may be limited by specifying the "`NUMREC`" parameter for `DAZUNDLI`. The syntax of this control card is as follows:

```
NUMREC=number_of_records
```

where *number_of_records* is the maximum number of unloaded records. This number can be up to 8 digits long.

Every occurrence of every segment counts as one record. Thus if the “number_of_records” is reached, it may be that not all dependent segments of the last unloaded root segment have been unloaded.

Example:

Unload at most 1000 records from the database:

```

NUMREC=1000
```

Limit the Unloaded Root Segment Occurrences

The number of unloaded root segment occurrences may be limited by specifying the "`NUMROOT`" parameter for `DAZUNDLI`. The syntax of this control card is as follows:

```
NUMROOT=number_of_roots
```

where *number_of_roots* is the maximum number of unloaded root segment occurrences. This number can be up to 8 digits long.

If the `NUMREC` parameter is not specified, the root segments will be unloaded together with all their dependent segments.

Example:

Unload at most 50 root segment occurrences together with their dependents:

```
NUMROOT=50
```

Unload a Specific Range of Values

The range of unloaded root segment occurrences may be limited to specific values by defining the "`START`" or "`END`" parameter for `DAZUNDLI`. The syntax of these control cards is as follows:

```
START=string
END=string
```

where *string* is either the start or end value and must be of the following format:

```
'char_string'
X'hex_string'
```

where

char_string	may contain any character
hex_string	must consist of pairs of the characters 0-9, A-F, where each pair will be interpreted as a hexadecimal character.

A string may be continued by ending the current line with a comma (“,”). The following line must contain only a string without any keyword being specified. This continued string may start in any column, may itself be continued and may have either format. Thus, it is possible to build up a start and end value from both character and hexadecimal strings.

The database will be unloaded from the start to the end value, inclusively. The end value might not be reached if the `NUMREC` or `NUMROOT` parameter is specified.

The values refer to the root sequence field or, if an alternate sequence is chosen, to the secondary index field. If the string is longer than the referenced field, it is truncated at the right. If it is shorter, it is padded at the right with low values (`hexadecimal X'00'`) or high values (`X'FF'`) for the start or end value respectively.

Packed values may be specified in hexadecimal format. It is recommended to use the correct field length, as this avoids an undesired truncation or padding of the packed values.

The following example shows how to unload data for the root sequence field range from "`EDV`" to "`MATH`" inclusively:

```
START='EDV'
END='MATH'
```

The following example shows how to unload all data with a root sequence field value of "XYZ" followed by a hexadecimal “01”:

```
START='XYZ',
  X'01'
END='XYZ',
  X'01'
```

Unloading Specific Values

Specific root segment occurrences may be unloaded by specifying the ROOTKEYS parameter. The syntax of this control card is as follows:

```
ROOTKEYS
key values
```

or

```
ROOTKEYS=SEQ
```

In the first case, the ROOTKEYS must be the last parameter for DAZUNDLI. It is followed by one or more root sequence field values. If ROOTKEYS=SEQ is specified, the root sequence field values are read in from DAZIN3. In this case, it is not required that the ROOTKEYS parameter is the last parameter. The corresponding root occurrences are unloaded together with all dependents. When the ROOTKEYS parameter is specified, the START and END parameter must not be used.

The following example shows how to unload the data for the root sequence fields with the values 'EDV' and 'GERMAN'.

```
ROOTKEYS
EDV
GERMAN
```

Unloading a HDAM Database

In a HDAM database, the sequence of the root occurrences is defined by the randomizing module. This mostly does not correspond to the root key sequence. When such a database is unloaded (without additional parameters), the reloaded Adabas data is randomly distributed over the data blocks. This results in a poor performance for sequential reads, since for each new accessed root segment occurrence, a physical I/O is required.

Thus it is recommended to unload the data in root key sequence. To do this, use any application which writes all the root sequence field values (and only these) to a sequential file. Sort these values by using any sort utility. These sorted values can now be used as key values for the `ROOTKEYS` parameter of the `DAZUNDLI` utility as described above. This forces DL/I to unload the database in the given sequence and the loaded Adabas data is no longer randomly distributed over the data blocks.

If the data has been initially loaded in the randomized sequence, i.e., without the `ROOTKEYS` parameter, it can be sequenced by unloading and reloading it from the ADL files. This time the `ROOTKEYS` parameter is not required, because ADL uses the root sequence field always as sort key.

Control Statements for the ADL Unload Utility

The following keywords are available for `DAZUNDLI`. For a detailed description refer the previous sections. The keywords are read from the control input for `DAZUNDLI` (see the sections *z/OS JCL Requirements*or *z/VSE JCS Requirements* later in this documentation).

Keyword	Explanation	Possible values	Default
END	Indicates the end value of the unload. The value refers to the root sequence field, or if an alternate processing sequence is specified, the corresponding secondary index field. The value may be continued by putting a comma after it.	Any character string or an “X” followed by pairs of the characters 0-9, A-F, which will be interpreted as hexadecimal characters. Both types of string have to be enclosed by quotation marks.	None. If no other condition is met, the database is unloaded until the end of the database is reached.
FIELD	The name of a numeric field to be checked for valid values. This keyword must be preceded by the SEGM keyword.	Any numeric field name.	None.
MODE	Indicates, whether numeric fields are to be checked for valid values. The MODE keyword can be overridden by the SEGM keyword.	STANDARD - do not check numeric fields. CHECKNUM - check all numeric fields.	
NUMREC	The maximal number of unloaded records.	1 - 99999999	No limit.
NUMROOT	The maximal number of unloaded root segment occurrences.	1 - 99999999	No limit.
ROOTKEYS	Unload specific root key values together with their dependents.	None - the key values are supplied after the ROOTKEYS parameter. SEQ - the key values are in DAZIN3.	If ROOTKEYS is not specified at all, all root key values are unloaded.
SEGM	The name of a segment with a numeric field to be checked for valid values. This keyword must be followed by the FIELD	Any segment name.	None.

Keyword	Explanation	Possible values	Default
	keyword. Once the SEGM keyword is specified, only the corresponding fields will be checked, regardless of the MODE keyword.		
START	Indicates the start value of the unload. The value refers to the root sequence field, or if an alternate processing sequence is specified, the corresponding secondary index field. The value may be continued by putting a comma after it.	Any character string or an "X" followed by pairs of the characters 0-9, A-F, which will be interpreted as hexadecimal characters. Both types of string have to be enclosed by quotation marks.	None.

Data Unload With the ADL Customized Utility

To use the manual conversion method to unload the data stored in a DL/I database, perform the following steps:

- Step 1: Convert the Physical DBD
- Step 2: Create a DL/I Unload PSB
- Step 3: Create an ADL Reformat PSB
- Step 4: Unload Data
- ADL Request Handler (DAZBRQH)
- Step 5: Reformat the Data

Step 1: Convert the Physical DBD

Run the original DL/I DBD through the CBC utility (see the topic *ADL Conversion Utilities for DBDs and PSBs* in this section).

Step 2: Create a DL/I Unload PSB

The PSB created must contain one PCB: this must be based on the original DL/I DBD and must reference all segments of the DBD. Run the PSB through the DL/I `PSBGEN`. Note that if the DBD is involved in logical relationships and contains logical child segments, these must also be referenced by the PCB (this also applies to virtual logical child segments).

Step 3: Create an ADL Reformat PSB

Run the PSB created in Step 2 through the CBC utility (see the topic *ADL Conversion Utilities for DBDs and PSBs* in this section).

Step 4: Unload Data

Write, assemble and link edit an unload program to unload the data from the DL/I database. An example of an unload program is provided as the source member `DAZUNLOD` (z/OS) or `DAZUNLOD.A` (z/VSE) in the Source Library on the installation tape.

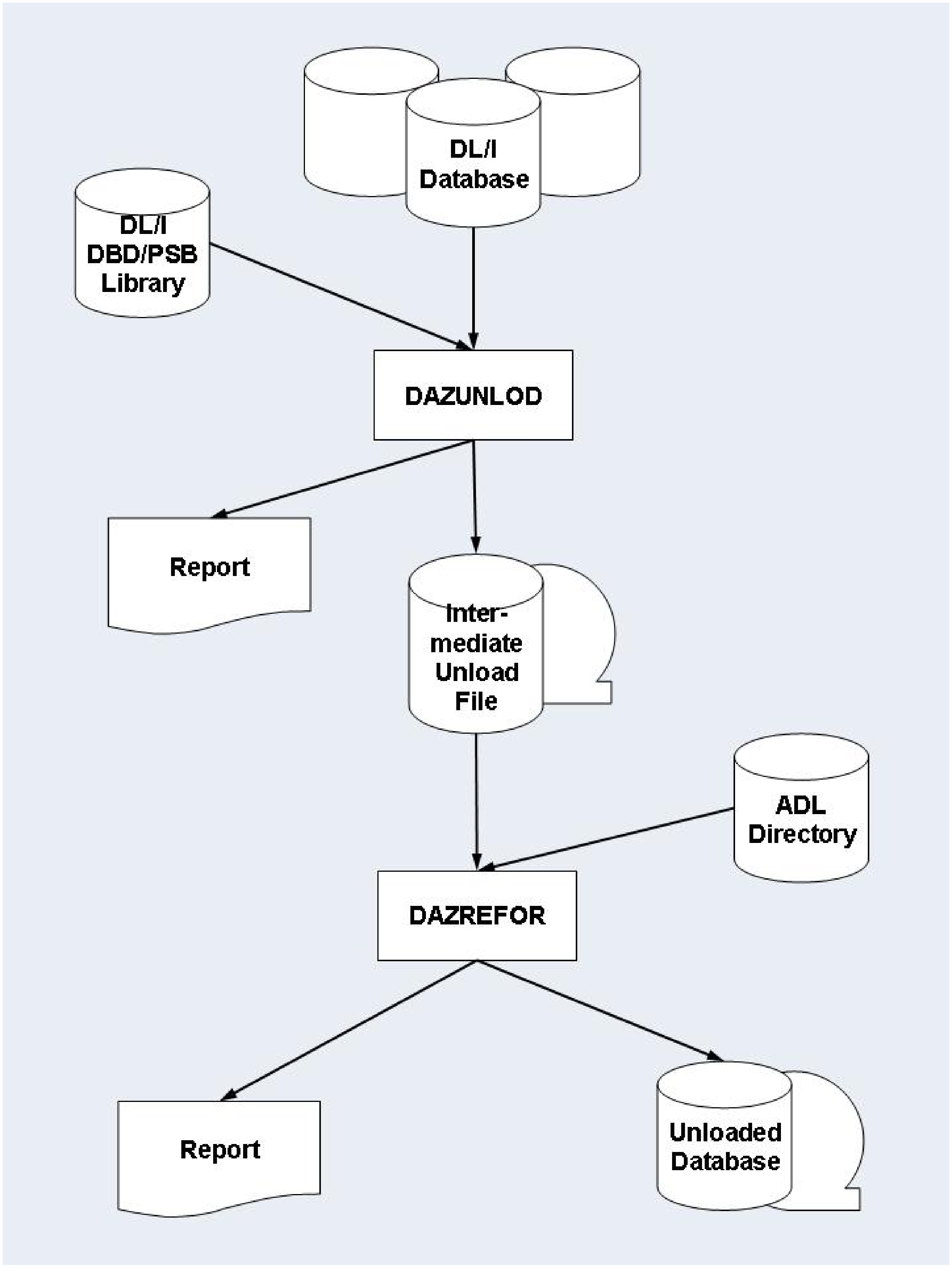
Write the unload program in Assembler using the standard programming conventions for this language.

The unload program reads the DL/I database by issuing unqualified `GN` calls until it reaches the end. If the standardized reformat program `DAZREFOR` is to be used in Step 5 (below) to create the input file for `ADACMP`, the segment occurrences read have to be written out to the intermediate unload file as variable length records with the following record layout:

Bytes	Explanation
1-4	Length of the record, including this field (standard convention for variable length record). (4 bytes)
5-12	The DL/I segment name taken from the SENSEG feedback area in the user PCB. (8 bytes)
13-	The DL/I segment data taken from the I/O area. Note that if the segment is of variable length, the length bytes are part of the segment data and must be included as the first two bytes in the segment I/O area.

The unload program is executed as a normal DL/I application program using the DL/I unload PSB.

Data Base Conversion Unload (Manual Procedure)


DL/I Database
DL/I DBD/PSB Library
DAZUNLOD
Report
Inter-mediate Unload File
ADL Directory
DAZREFOR
Report
Unloaded Database

ADL Request Handler (DAZBRQH)

When using the sample program on the installation tape, you should bear in mind that it uses the ADL internal request handler, `DAZBRQH`. It is therefore independent of the operating system used.

When `DAZBRQH` is being used, certain conventions need to be followed. These are described in detail in the example program itself, but are summarized briefly below.

Save and Work Areas

Save and work areas for the request handler must be provided by the caller. The first fullword of this area must contain the start address of the area itself, i.e. it must be pointing to itself. In the sample program, this is achieved by using the macro `SAVEAREA`. For z/VSE users, this macro may also be used to change the block size of the intermediate unload file dynamically.

Requests

Requests are issued by calling the request handler as a subroutine, with the registers `R0` and `R1` being used to pass the necessary parameters. In the sample program, this is achieved by using the `REQUEST` macro.

Files

All files are opened by the request handler dynamically, but have to be closed by the caller.

Sample Program/`DAZBRQH`

Because the sample program uses the internal request handler, it must be linked with `DAZBRQH`.

Step 5: Reformat the Data

The intermediate unload file created in Step 4 is read and each unloaded segment is put through ADL. The input file for `ADACMP` is created. The standardized reformat program in the Load Library, `DAZREFOR` may be used if the intermediate unload file has been created according to the layout conventions in Step 4.

If the standardized reformat program does not meets your requirements, you can write your reformat program. An example of a reformat program is provided as the source member `DAZREFOR` (z/OS) or `DAZREFOR.A` (z/VSE) in the Source Library.

The reformat program reads the intermediate unload file and issues a load call against ADL for each unloaded segment occurrence. A load call has the same syntax as a normal DL/I insert call, but `LOAD` is specified as the function instead of `ISRT`.

The parameters for the `LOAD` call are as follows:

Parameter	Explanation
Parameter 1	(optional) The number of parameters (4) that follow.
Parameter 2	The function is LOAD.
Parameter 3	The PCB address for the first and only PCB in the reformat PSB.
Parameter 4	The I/O area as read from the intermediate unload file.
Parameter 5	An unqualified SSA specifying the name of the segment as read from the intermediate unload file.

The standardized reformat program `DAZREFOR` is executed as a normal mode ADL program (see the section Batch Installation and Operation in the *ADL Interfaces* documentation). The execution parameters must have the following layout:

```
REF,DAZREFOR,psbname
```

where *psbname* is the name of the ADL reformat PSB.

Like `DAZUNLOD`, `DAZREFOR` uses the ADL internal request handler, `DAZBRQH`. The same conventions apply.

Converting Data - Load

The manual and the automated procedures use the same methods of loading Adabas files. You must perform the following steps:

* Only for databases related via a bidirectional logical relationship and if the standard or simplified procedure of `DAZELORE` is used to establish logical relationships.

This section covers the following topics:

Step 6: Initial Load of the Adabas File(s)

Each Adabas file used to store the converted data is loaded individually using the standard Adabas utilities (`ADACMP, ADALOD`).

The sequential file produced in the previous steps is taken as input for the `ADACMP` step, as are the `ADACMP` statements generated by the CBC utility in Step 1.

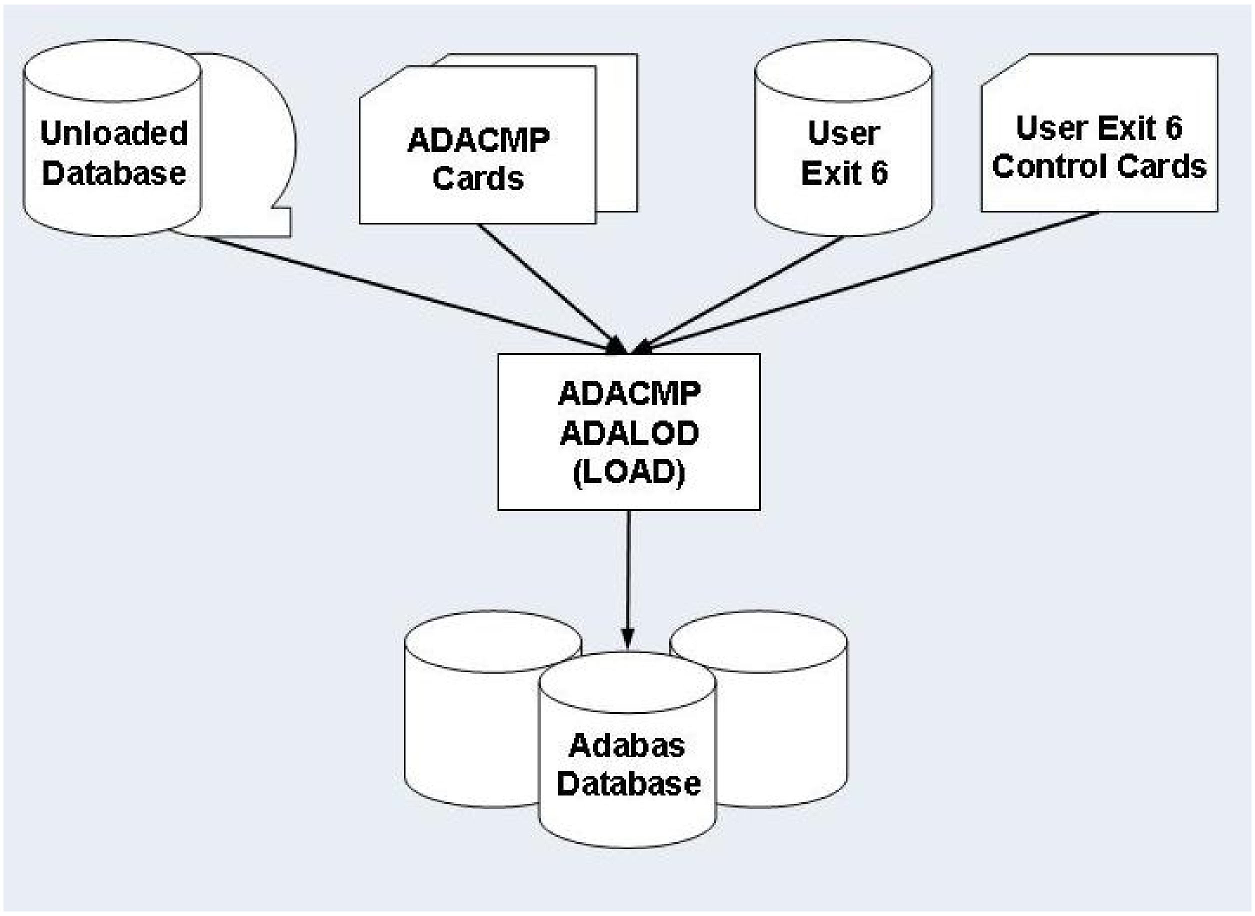
Each Adabas file used must be loaded with the option `USERISN`. This applies to both the `ADACMP` and the `ADALOD` steps (for `ADACMP` it is already generated by the CBC utility), as the ISNs are automatically generated by the ADL.

Step 6 needs to be run once for each of the Adabas files used to store the DL/I database.

Note that the initial load does not load the data of a paired logical child segment of a bidirectional logical relationship. This is performed by the next step, if required at all.

You may use the sample JCL (JCS) in the member `ADLDBC6` (`ADLDBC6.J`) as an example of an initial load job under z/OS (z/VSE).

Data Base Conversion Initial Load



Adabas User Exit 6

The ADACMP step uses Adabas User Exit 6. This exit consists of two parts which were linked together during the DBD conversion procedure (see the section **ADL Conversion Utilities for DBDs and PSBs** in this documentation):

1. *Fixed Part*

   The fixed part consists of the DAZUEX06 module, which expands each input record on the unload file to its full decompressed size before passing it on to ADACMP.

2. *User Exit 6 Extension*

   The User Exit 6 extension is generated by the CBC utility and contains information on the structure of the DBD being converted, and the default record layouts of the Adabas file(s) used to store the converted data.

Adabas User Exit 6 needs a control statement to indicate which Adabas file should be loaded. The syntax of this control statement is as follows:

```
FNR=nnnnn,MODE=LOAD
```

where *nnnnn* is the file number of the Adabas file to be loaded.

Information on how to load the file is provided at the end of this section.

Step 7: Mass Update for Paired Logical Child Segments

You only need to perform this step if the database being converted is involved in bidirectional logical relationships. In this case, the data originating from a bidirectional logical segment is stored only once, in an Adabas file. The way in which the data was stored under DL/I (virtual or physical pairing) has no effect on this.

You must not perform this step when the Special or Turbo procedure is used to establish logical relationships (see the following section: *Establishing Logical Relationships*).

Within a bidirectional logical relationship, the real logical child segment can be defined as the logical child segment for which an Adabas file number and Adabas fields are generated. The paired logical child segment can be defined as the logical child segment with which the real logical child segment is paired. The reports produced by the CBC utility during conversion of the physical DBDs show which Adabas file contains the logical child segment data, and therefore which of the two logical child segments is the real logical child segment. Once all Adabas files storing data originally contained in DL/I databases related via a bidirectional logical relationship have been loaded, the Adabas Mass Update utility `ADALOD` with the `UPDATE` function must be run for all Adabas files which contain data originating from a bidirectional logical child segment.

The real logical child segment occurrences are unloaded during the unload of the DL/I database in which they were stored. They are subsequently loaded with an initial load using the Adabas utilities `ADACMP` and `ADALOD`. The paired logical child segment occurrences are unloaded separately during the unload of the DL/I database in which they were stored and must be added separately to the Adabas file in which the real logical child segment occurrences have been loaded. This is done using `ADALOD` (`UPDATE`). To produce the input for `ADALOD` (`UPDATE`), run the Adabas Compression utility `ADACMP` with the following input:

- The `ADACMP` statements for the Adabas file used to store the bidirectional logical child segment data, i.e. the Adabas file for the *real* logical child segment. Note that because the output produced by the `ADACMP` utility is input to the `ADALOD` (`UPDATE`) utility, the `USERISN` option must not be specified and has to be removed from the `ADACMP` statements before you run the `ADACMP` step.
- The Adabas User Exit 6 for the Adabas file used to store the bidirectional logical child segment data, i.e. the Adabas file for the *real* logical child segment.
- The unloaded DL/I database containing the *paired* logical child segment data.
- A User Exit 6 control statement specifying a mass update run with the format:

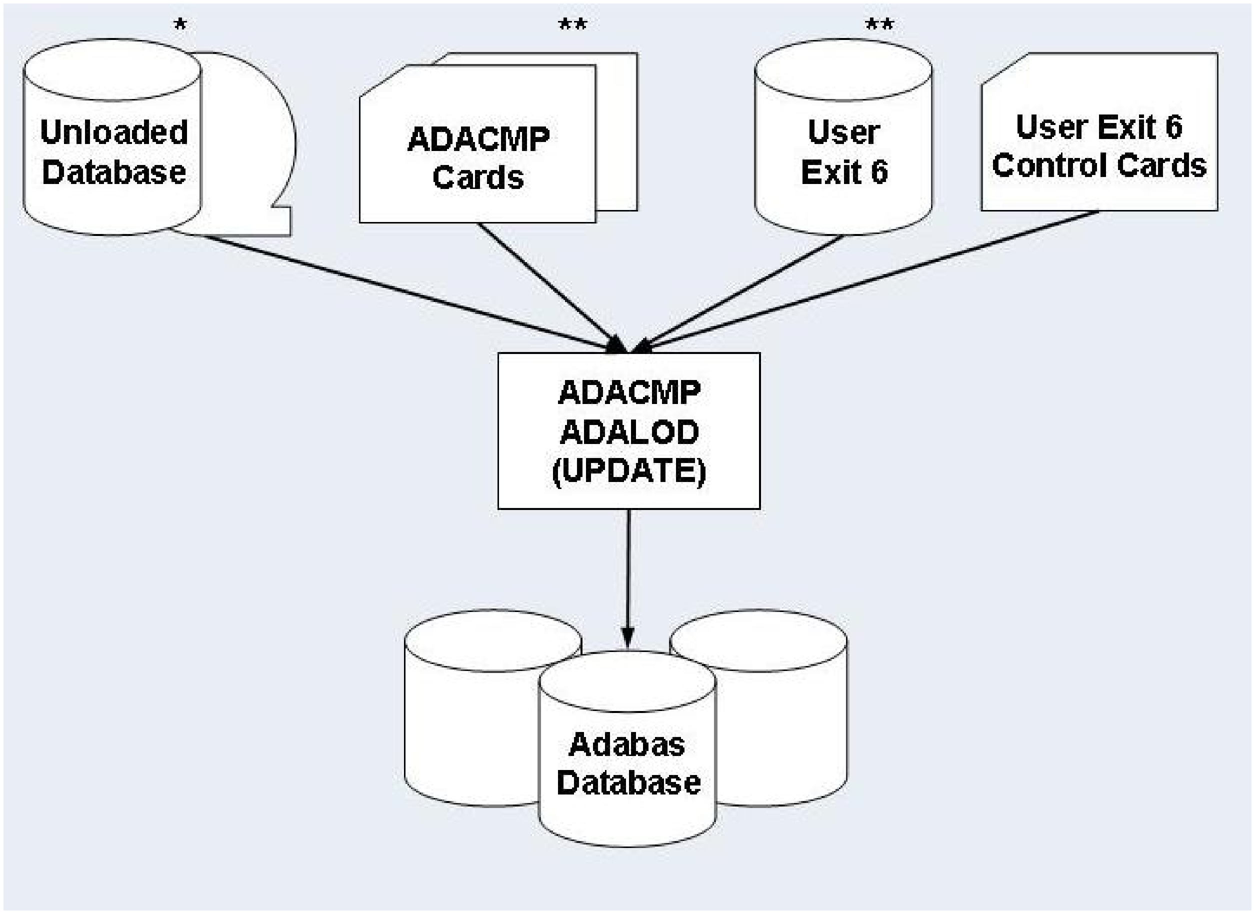
```
FNR=nnnnn,MODE=MASS,LC=name
```

where

nnnnn	is the file number of the Adabas file containing the bidirectional logical child segment data, i.e. the Adabas file for the real logical child segment, and
name	is the name of the paired logical child segment.

You may use the sample JCL (JCS) in the member `ADLDBC7` (`ADLDBC7.J`) as an example of a mass update under z/OS (z/VSE).

Data Base Conversion Mass Update



* Unloaded Database containing the *paired* logical child segment data.

** ADACMP Cards and User Exit 6 generated for the *real* logical child segment.

Establishing Logical Relationships

When the first seven unload/load steps have been successfully completed, the data stored in the DL/I database have been converted to one or more Adabas file(s). Where no logical relationships exist for a DBD, no other steps are necessary. However, where one or more logical relationships exist, each logical child segment occurrence has to be "connected" to its logical parent. This is done in two steps using the `DAZELORE` (Establish Logical Relationship) utility.

Step	Description
Step 8	Create connect PSB.
Step 9	Establish logical relationship.

Four different procedures for Steps 8 and 9 - Standard, Simplified, Special and Turbo - exist (see below). The procedure you should use in any given case depends on the DBDs and user applications involved. All procedures involve creating a connect PSB and executing `DAZELORE`; the difference is that the Standard procedure requires both the original DL/I and the converted databases, while the other procedures only require the latter. The Standard procedure is thus more time-consuming but can be used in all cases. In contrast, the Simplified, Special and Turbo procedures can only be used for converted databases in which no logical child segments without matching logical parents and no variable intersection data segments which are no longer accessible via their physical parents exist. The Special procedure differs from the Simplified procedure only for bidirectional logical relationships. It can only be used if every logical child segment occurrence has a paired logical child segment occurrence present in the database. For the Turbo procedure the pre-requisites of the Special procedure must be fulfilled. Additionally all parent segments on the logical and physical path up to and including the root segments must have unique sequence fields.

Steps 8 and 9 have to be performed separately for every logical child: i.e. once for every unidirectional logical relationship and twice for every bidirectional logical relationship for both the Standard and the Simplified procedures. The Special and Turbo procedure, however, only needs to be run once per logical relationship, regardless of whether or not this relationship is bidirectional.

When large databases are being converted, performance problems may arise with these steps. We therefore recommend that you run the `DAZELORE` utility in single user mode where possible. In addition, you should use the Turbo, Special or Simplified methods wherever possible as these procedures do not require that the original DL/I database be accessed simultaneously. The highest performance is provided by the Turbo procedure.

Furthermore, the `DAZELORE` utility may be run using checkpoints, which means that a particular run may be split up into several sub-runs if necessary. See the section entitled *Restart Considerations* later on in this section for details of how to use checkpoints with the `DAZELORE` utility and how to perform restarts.

This section covers the following topics:

- Standard Procedure
- Simplified and Special Procedures
- Turbo Procedure
- DAZELORE Run Report

- Restart Considerations

Standard Procedure

The Standard procedure is required in all cases in which segment occurrences which were originally present in the DL/I database have been physically deleted, but are still accessible via a logical path. This may occur in one of two situations:

- With segment occurrences which have been physically deleted but which are still accessible via a logical child segment.

The physical unload performed in the unload procedure does not unload these physically deleted segment occurrences. Logical child segment occurrences may thus be present in the converted database(s), even though no matching logical parent segment occurrences exist (even the parents of these logical parents may be missing). The missing segment occurrences have to be added to the Adabas files in such a way as to be accessible via logical, not physical, paths.

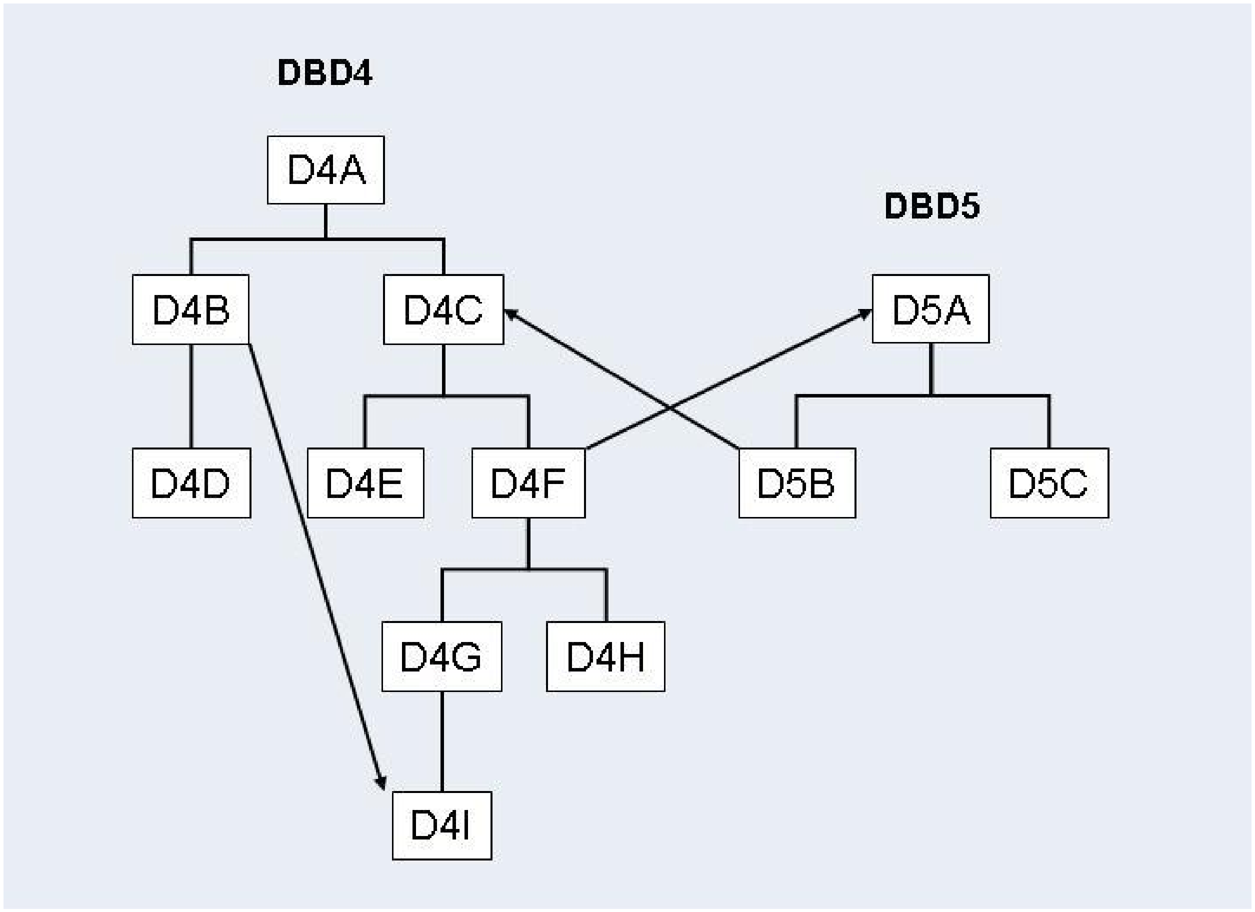
- Segment occurrences which are variable intersection data may no longer be accessible via their physical parents but may still be accessible via a logical child segment.

Again, the physical unload performed in the unload procedure does not unload these segment occurrences. The missing segment occurrences have to be added to the Adabas files in such a way that they are only accessible via the logical child.

All missing segments are reinserted during the `DAZELORE` runs using the Standard procedure.

The two situations listed above are explained in the set of examples following.

Physical DBDs and their Logical Relationships



In the example above, the two DBDs, `DBD4` and `DBD5`, are involved in two logical relationships:

- A unidirectional logical relationship between the logical child segment `D4B` and the logical parent segment `D4I` within `DBD4`;
- A bidirectional logical relationship between `DBD4` and `DBD5` with the logical child segments `D4F` and `D5B` and the logical parent segments `D4C` and `D5A`.

Certain segments may be physically deleted but still logically accessible. This depends on the setting of the delete rules for the segments involved in the logical relationships and on the logic of the application programs.

Segments `D4A`, `D4C`, `D4F`, `D4G`, and `D4I` may have been physically deleted but may still be accessible via the logical child segment `D4B`. The segment occurrences will not have been unloaded with `DBD4`, and will need to be inserted during the `DAZELORE` run for `D4B`.

Segments `D4A` and `D4C` may have been physically deleted but may still be accessible via the logical child segment `D5B`. The segment occurrences will not have been unloaded with `DBD4`, and will need to be inserted during the `DAZELORE` run for `D5B`.

Segment D5A may have been physically deleted but may still be accessible via the logical child segment D4F. The segment occurrences will not have been unloaded with DBD5, and will need to be inserted during the DAZELORE run for D4F.

In addition, segments D4G, D4H and D4I may no longer be accessible via their physical parent D4F, but may still be accessible via D5B. The segment occurrences will not have been unloaded with DBD4, and will need to be inserted during the DAZELORE run for D5B.

Step 8: Creating a Standard Connect PSB

The Standard connect PSB must contain four PCBs: PCB1, PCB2, PCB3 and PCB4. All these PCBs must specify processing option "A". PCB1 and PCB3 must also specify processing option "P" for path calls.

PCB1

PCB1 is based on the converted DBD and references the logical child segment and all its parent segments only. The PCB must be based on the physical DBD. For this reason, the sensitive segment (SENSEG) describing the logical child segment cannot describe the concatenated segment.

PCB2

PCB2 is based on the converted DBD and references the logical child segment, all its parent segments, and all the parent segments of the logical parent in the inverted structure. The PCB has to be based on the logical DBD. The SENSEG describing the logical child segment must describe the concatenated segment. Where the logical child segment is a bidirectional logical child and the paired logical child segment has dependents (i.e. variable intersection data segments), the latter must be included as dependents of the logical child as well.

PCB3

PCB3 is based on the converted DBD and references the logical parent segment and all its parent segments only. The PCB has to be based on the physical DBD.

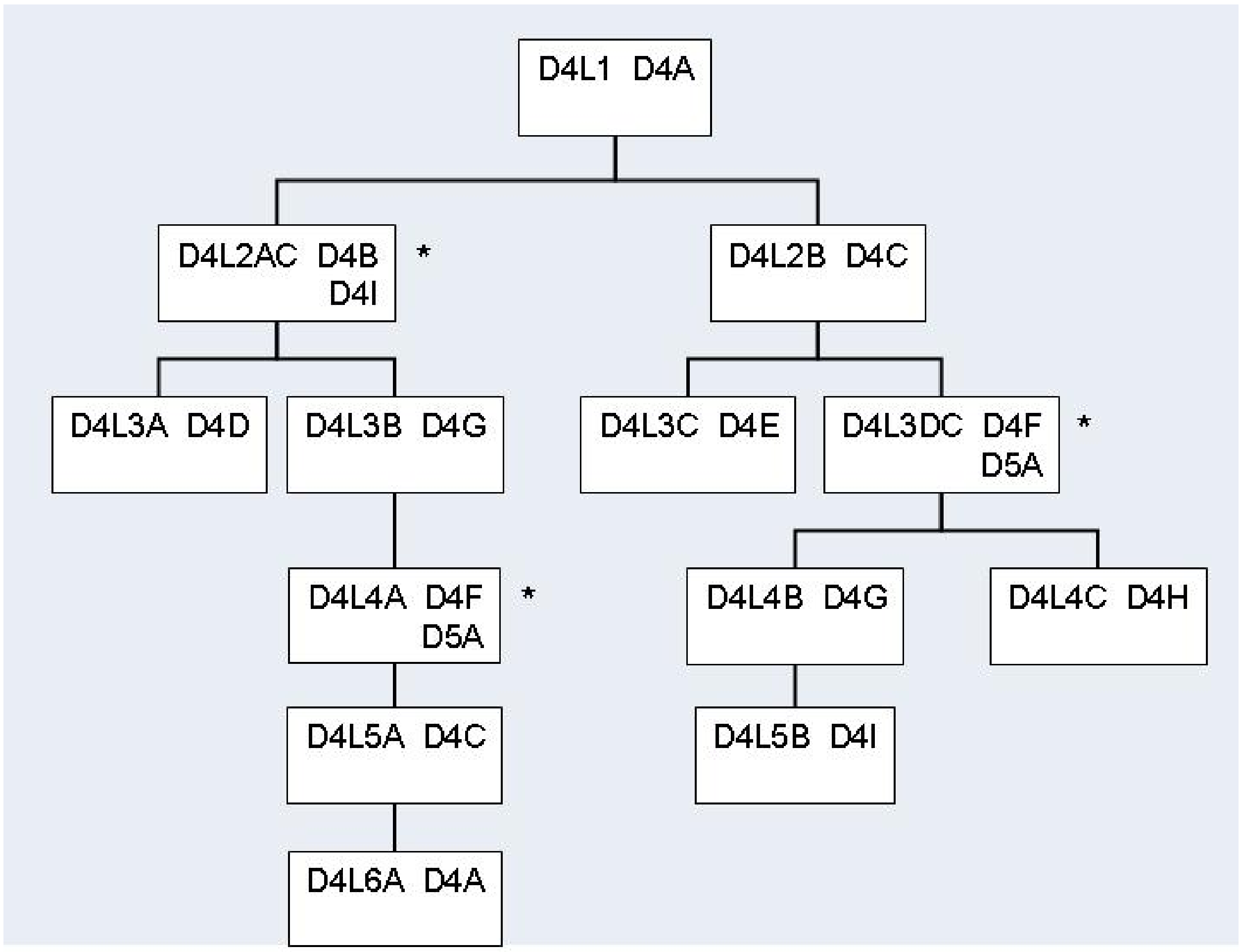
PCB4

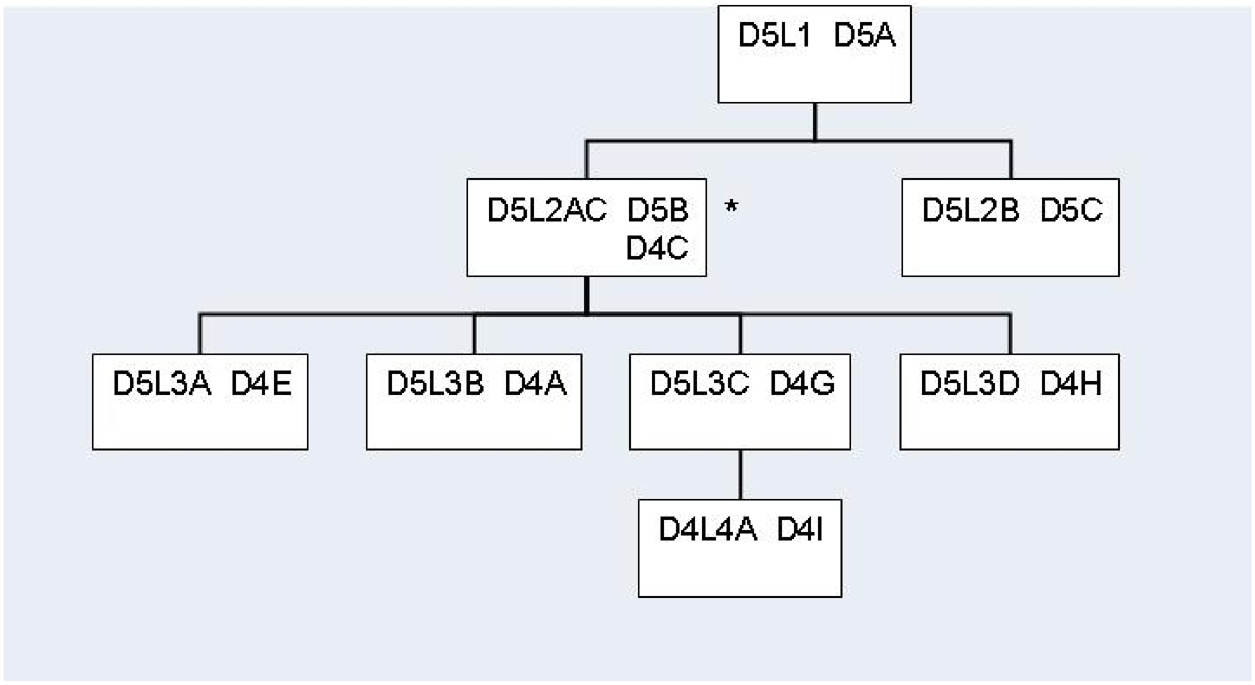
PCB4 is only needed where the logical child segment is a bidirectional logical child and its paired logical child segment has dependents. In all other cases it may be omitted. PCB4 is based on the converted DBD and references the logical child segment and all its parent segments. It must also reference all variable intersection data segments as dependents of the logical child. The PCB must be based on the logical DBD. The SENSEG describing the logical child segment must describe the concatenated segment.

The Standard connect PSB must be run through the DL/I PSBGEN and the CBC utility (see the section *ADL Conversion Utilities for DBDs and PSBs* in this documentation).

The following figures illustrate this in more detail.

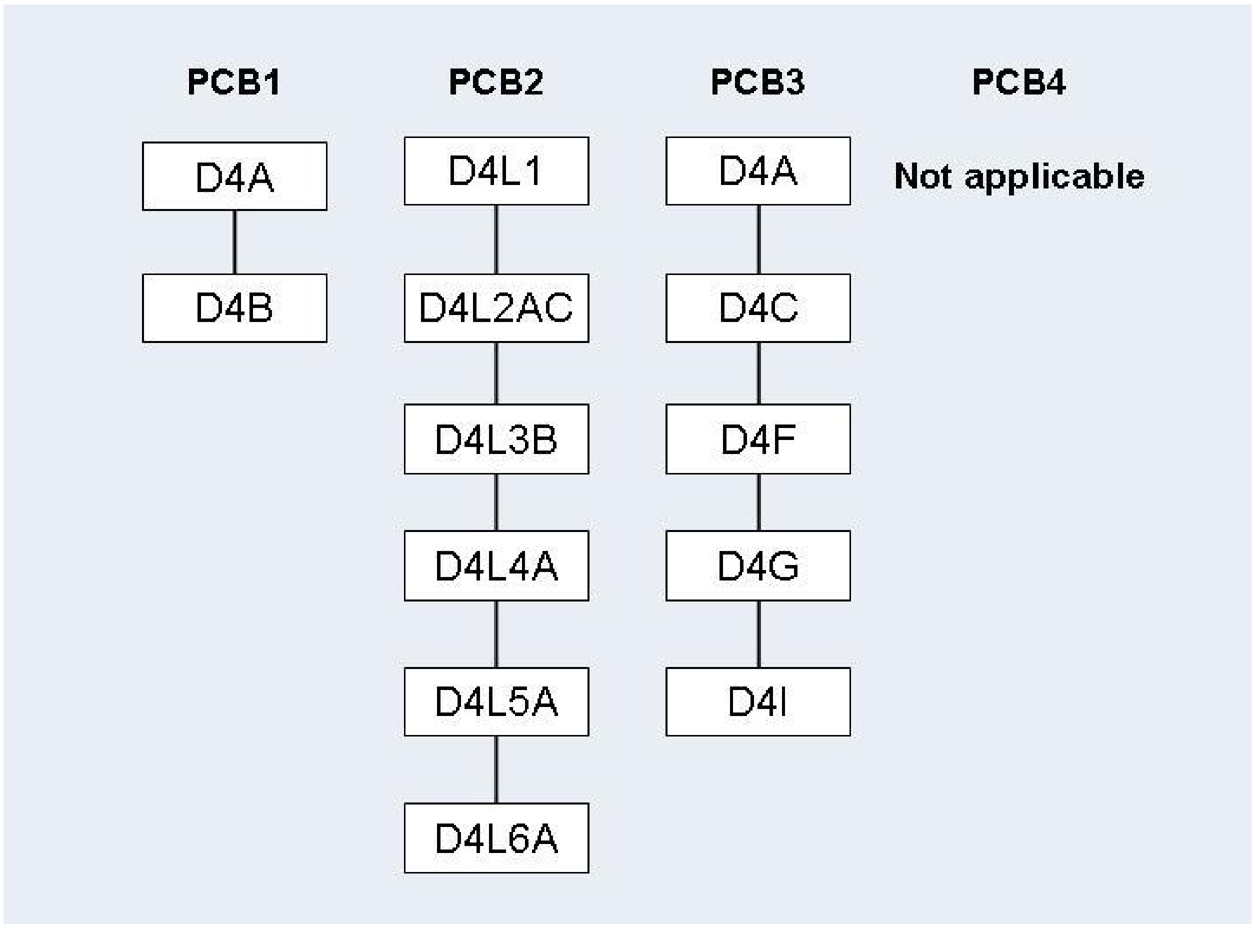
Logical DBDs based on DBD4 and DBD5


D4L1 D4A
D4L2AC D4B D4I
*
D4L2B D4C
D4L3A D4D
D4L3B D4G
D4L3C D4E
D4L3DC D4F D5A
*
D4L4A D4F D5A
*
D4L4B D4G
D4L4C D4H
D4L5A D4C
D4L5B D4I
D4L6A D4A

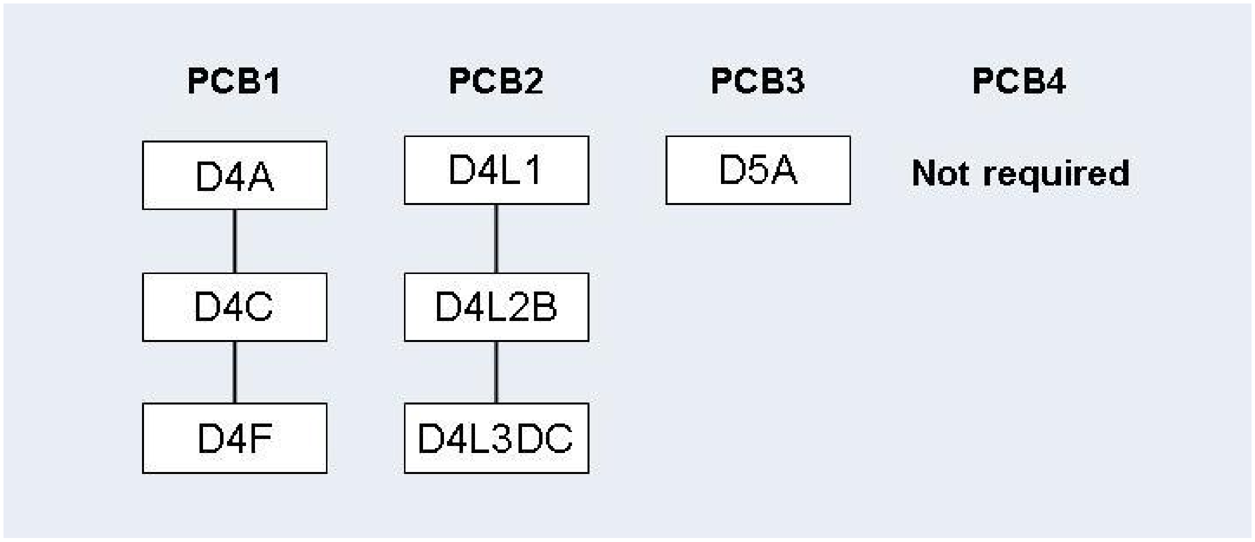

D5L1 D5A
D5L2AC D5B D4C
*
D5L2B D5C
D5L3A D4E
D5L3B D4A
D5L3C D4G
D5L3D D4H
D4L4A D4I

* concatenated segments

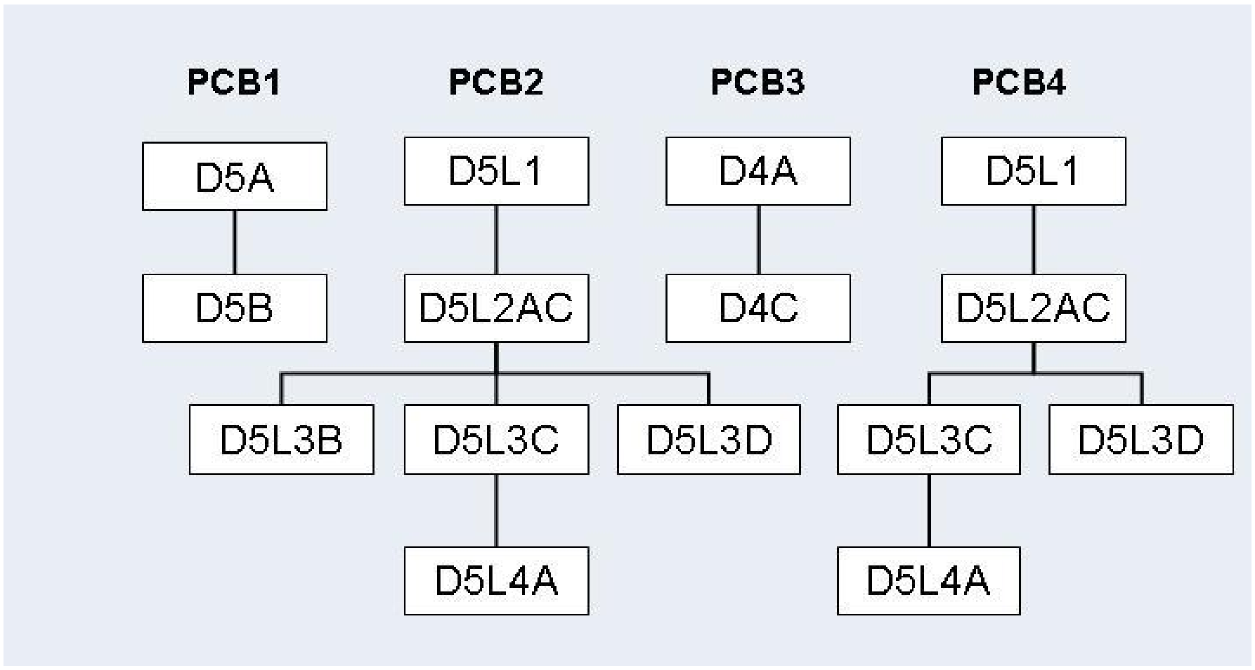
Standard Connect PSB for Logical Child D4B



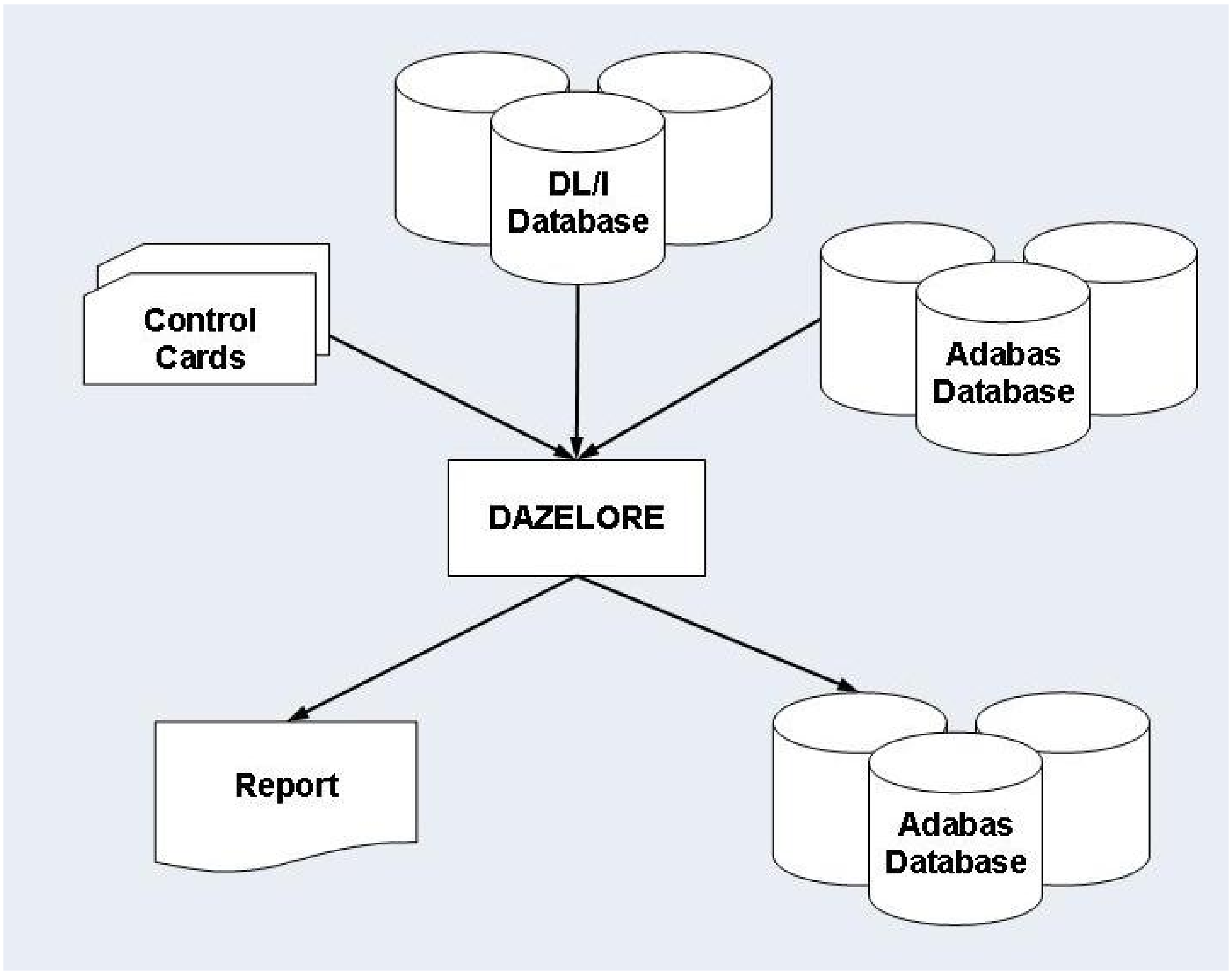
Standard Connect PSB for Logical Child D4F



Standard Connect PSB for Logical Child D5B



Establishing a Logical Relationship (Standard Procedure)



Step 9: Establishing Logical Relationship - Standard Procedure

Logical children and parents are connected by running the ADL Establish Logical Relationships utility, `DAZELORE`. In the Standard procedure, `DAZELORE` is executed as a mixed mode program (see the section *Batch Installation and Operation* in the *ADL Interfaces* documentation) using a mixed mode control statement with the following layout:

```
ELO,DAZELORE,psbname
```

where *psbname* is the name of the Standard connect PSB.

`DAZELORE` also needs a control statement to indicate which logical child segment should be connected. The syntax of this control statement must be as follows:

```
LC=lognam,MODE=STANDARD
```

where *lognam* is the name of the logical child segment in the physical DBD.

The specification of the `LC` parameter is mandatory and it must be the first parameter specified.

Where a DBD is involved in more than one logical relationship, several `DAZELORE` jobs need to be run. The order for this is only important where variable intersection data segments exist and at least one of them is a logical parent. In this case, the `DAZELORE` run which may cause variable intersection data segments to be inserted, must be run before the `DAZELORE` runs connecting the logical child segments to the variable intersection data segments which are logical parent segments.

In the case of the examples given on the previous pages, the order in which the jobs have to be run is as follows:

1. *The DAZELORE run for segment D5B.*

   This is because segment `D4I` is both a variable intersection data segment and a logical parent, and during this run `D4I` segment occurrences may be inserted.
2. The order of the two remaining `DAZELORE` runs (for `D4B` and `D4F`) is irrelevant.

Because segment occurrences may be inserted during `DAZELORE` runs, the situation may arise in which logical child segment occurrences are inserted although the `DAZELORE` run for the logical child segment has already been performed. To establish whether this is the case, look at the report which is printed out at the end of each `DAZELORE` run and which gives all the segment occurrences inserted during that run. If a segment occurrence has been inserted, rerun the `DAZELORE` utility for that logical child segment.

For example, let us assume that, in the illustrations given previously, the `DAZELORE` runs for `D5B` and `D4F` have been successfully performed. The `DAZELORE` run for `D4B` may have triggered the insertion of `D4F` segment occurrences. If this is the case, the `DAZELORE` run for `D4F` must be repeated.

Simplified and Special Procedures

The Simplified and Special procedures may be used in all cases in which the original DL/I database did *NOT* contain any segment occurrences which have since been physically deleted but which are still accessible via a logical path.

The Special procedure only differs from the Simplified procedure for bidirectional logical relationships. It can only be used if all logical child segment occurrences have a matching paired logical child occurrence and vice versa, i.e. if both logical access paths are always present for any logical child-logical parent link. This fact has to be ensured by the user, for example by checking whether the numbers of unloaded records of the paired segments in the `DAZUNDLI` report are equal. The special procedure creates the paired logical child segments in accordance with the information extracted from the real logical child segment. Note that for the special procedure the logical child segment for which `DAZELORE` is run has to be the *real* logical child segment.

The advantages of the Special procedure are that the Mass Update step is not required during loading of the data in Adabas files (see Step 7: *Mass Update for Paired Logical Child Segments*), and that only one `DAZELORE` run is needed to establish the bidirectional logical relationship. This single `DAZELORE` run also has certain performance advantages over that used in the Simplified procedure.

If a logical child segment occurrence which does not have a matching logical parent is encountered during the run, the following error message is produced.

```
ADL0612: Unexpected DP status code for DAZELORE procedure used
```

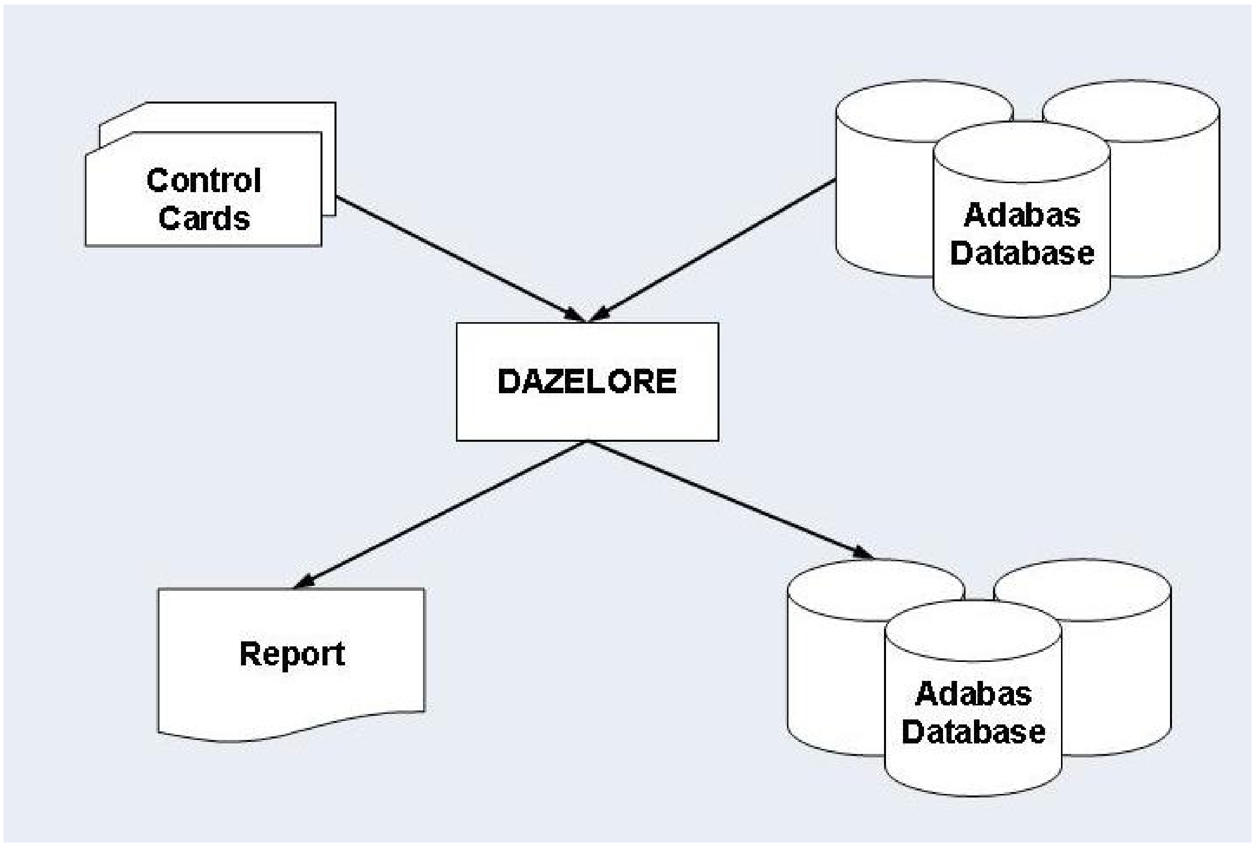
The job then terminates. In this case, rerun `DAZELORE` using the Standard procedure.

Step 8: Creating a Simplified or Special Connect PSB

Simplified and Special connect PSBs contain a single PCB based on the converted DBD and referencing the logical child segment and all its parent segments only. The PCB has to be based on the physical DBD. For this reason, the `SENSEG` describing the logical child segment cannot describe the concatenated segment. This PCB is identical to the PCB1 described in the section *Creating a Standard Connect PSB*. It must specify processing option "`AP`".

Simplified or Special connect PSBs must be run through the CBC utility (see the section *ADL Conversion Utilities for DBDs and PSBs* in this documentation).

Establishing a Logical Relationship (Simplified and Special Procedures)



Step 9: Establishing a Logical Relationship - Simplified or Special Procedure

This is done by running `DAZELORE`. In the Simplified and Special procedures, `DAZELORE` is executed as a normal mode program (see the section Batch Installation and Operation in the *ADL Interfaces* documentation). The control statement parameters must have the following layout:

```
ELO,DAZELORE,psbname
```

where *psbname* is the name of the Simplified or Special connect PSB.

`DAZELORE` also needs a control statement to indicate which logical child segment should be connected. The syntax of the control statement for the Simplified procedure is as follows:

```
LC=lognam,MODE=SIMPLIFIED
```

and that for the Special procedure is:

```
LC=lognam,MODE=SPECIAL
```

where *lognam* is the name of the logical child segment in the physical DBD.

You may use the sample JCL (JCS) in the member `ADLDBC9` (`ADLDBC9.J`) as an example of a `DAZELORE` run under z/OS (z/VSE).

Turbo Procedure

The Turbo procedure is the fastest way to build up the logical relationships. Whenever possible it is recommended to use the Turbo procedure.

The Turbo procedure can only be used if the following issues are satisfied:

- The pre-requisites of the Special Procedure are fulfilled. See the previous section for details.
- It can only be used for bi-directional logical relationships. Uni-directional relationships are currently not supported.
- All parent segments on the logical and physical path up to and including the root segments must have unique sequence fields.

Note that for the Turbo procedure the logical child segment for which `DAZELORE` is run has to be the *real* logical child segment.

Like the Special procedure, the Mass Update step is not required during loading of the data in Adabas files (see Step 7: *Mass Update for Paired Logical Child Segments*), and only one `DAZELORE` run is needed to establish the bidirectional logical relationship.

If a logical child segment occurrence which does not have a matching logical parent is encountered during the run, the following error message is produced.

```
ADL0612: Unexpected DP status code for DAZELORE procedure used
```

The job then terminates. In this case, rerun `DAZELORE` using the Standard procedure.

The Turbo procedure has the following performance advantages:

- It does not read the hierarchy to access the logical child segment data. Instead it uses the ADL internal pointer field to read the child segment data directly.
- The `ADARUN MULTIFETCH` feature can be used, when reading the logical child segments. It is recommended to use one `MULTIFETCH` buffer with maximum size which will contain the sequential reads of the logical child segment data.

- ADL (like DL/I) maintains for each logical child a counter at its physical parent and at its logical parent. The other procedures update these counters whenever a logical child is processed. The Turbo procedure updates the physical parent counter only once when all its children are processed. The logical parent counters are kept in an online table (“`DP counter table`”) and updated at the end of the run. If a counter in the DP counter table is bigger than 127, the corresponding counter in the database is updated and the counter is reset. Thus for every 128th logical child the counter of the logical parent is updated (and not for every logical child).
- The update of a logical parent counter in the sequence of the corresponding logical children (as done with the other procedures) is a “random” update and therefore very time consuming because usually every update requires a physical I/O. The Turbo procedure does not only collect the updates (as described before) but it makes also the final update in ISN sequence.

For a better performance it is recommended to use “`RESTART=NO`” with the Turbo procedure.

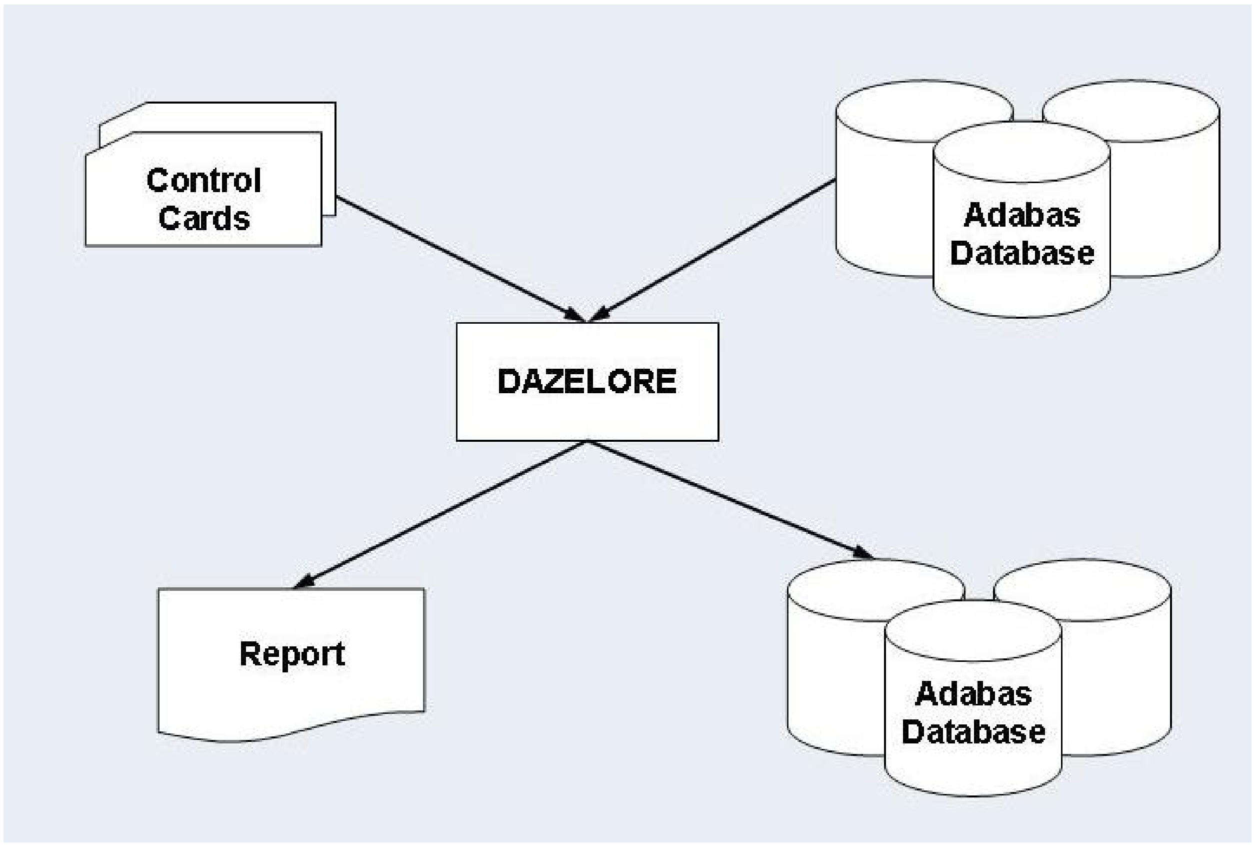
Step 8: Creating a Turbo Connect PSB

The Turbo connect PSB contains a single PCB based on the converted DBD and referencing the logical child segment and all its parent segments only (i.e. same as the Special connect PSB). The PCB has to be based on the physical DBD. For this reason, the `SENSEG` describing the logical child segment cannot describe the concatenated segment. This PCB is identical to the PCB1 described in the section *Creating a Standard Connect PSB*. It must specify processing option `"AP"`.

If `RESTART=YES` is specified for the `DAZELORE` run, the `KFB` (key-feedback area) length in the PCB must be at least 8 bytes long. If it is shorter, set it to 8 bytes.

The Turbo connect PSBs must be run through the CBC utility (see the topic *ADL Conversion Utilities for DBDs and PSBs* in this documentation).

Step 9: Establishing a Logical Relationship - Turbo Procedure



This is done by running `DAZELORE`. In the Turbo procedure, `DAZELORE` is executed as a normal mode program (see the section *Batch Installation and Operation* in the *ADL Interfaces* documentation). The control statement parameters must have the following layout

```
ELO,DAZELORE,psbname
```

where *psbname* is the name of the Turbo connect PSB.

`DAZELORE` also needs a control statement to indicate which logical child segment should be connected. The syntax of the control statement for the Turbo procedure is as follows:

```
LC=lognam,MODE=TURBO,MAXDPISN=n
```

where *lognam* is the name of the logical child segment in the physical DBD and *n* is the size of the DP counter table as described later in details.

You may use the sample JCL (JCS) in the member `ADLDBC9` (`ADLDBC9.J`) as an example of a `DAZELORE` run under z/OS (z/VSE).

DAZELORE Run Report

Each `DAZELORE` run produces a report. Such reports can be divided into two parts: a first part which is produced before any of the logical child segments are processed, and a second part which is produced at the end of a normally terminated run.

The first part of the report has the following layout and contains the following information:

```
CONNECT LOGICAL CHILD TO ITS LOGICAL PARENT
PROCEDURE........................: procedure
LOGICAL CHILD DBD................: LCDBD
LOGICAL CHILD SEGMENT............: LCseg
LOGICAL PARENT DBD...............: LPDBD
LOGICAL PARENT SEGMENT.......... : LPseg
```

where

procedure	is the procedure (Standard, Simplified or Special) used.
LCDBD	is the name of the DBD of the logical child segment.
LCseg	is the name of the logical child segment.
LPDBD	is the name of the DBD of the logical parent segment.
LPseg	is the name of the logical parent segment.

The second part of the report that is produced at the end of a successful run can have a variety of layouts and contain a variety of information.

If, during the processing of a logical child segment, `DAZELORE` finds that the destination parent segment has been physically deleted and subsequently reinserted, the following message is produced:

```
PHYSICALLY DELETED SEGMENTS HAVE BEEN REINSERTED
```

This message is followed by a list of all the segment types and the number of segment occurrences inserted, as shown below.

```
SEGMENT  QUANTITY
-------  --------
.......  ........
.......  ........
         --------
         ........
```

Alternatively, if no such situation has been encountered, the following message is produced:

```
NO PHYSICALLY DELETED SEGMENTS HAVE BEEN FOUND
```

Where a bidirectional logical child is being processed and the paired logical child segment has dependents (variable intersection data), variable intersection data segments may have been inserted during the run. In this case the following message is produced:

```
VARIABLE INTERSECTION DATA SEGMENTS HAVE BEEN INSERTED
```

It is followed by a list of segment types and quantities similar to that mentioned above.

Where no such segments were encountered, the following message is produced:

```
NO VARIABLE INTERSECTION DATA SEGMENTS HAVE BEEN INSERTED
```

Where a segment occurrence that is a logical child segment was inserted during the `DAZELORE` run, the following eye catcher is printed behind the segment name and quantity, to indicate that `DAZELORE` needs to be rerun for this segment type.

```
LOGICAL CHILD RE-INSERTED, RE-RUN DAZELORE
```

The following five messages appear at the end of the report:

```
THIS RUN PROCESSED .......NO....... LOGICAL CHILDREN
```

This message states the total number of logical child segment occurrences processed in the run. It is followed by the next message:

```
OF WHICH .......NO....... LOGICAL CHILDREN WERE ALREADY CONNECTED
```

which states the total number of logical child segment occurrences found to have been already processed in previous runs.

The third message states the total number of destination parent segment occurrences found to have been physically deleted.

```
.......NO....... DESTINATION PARENT SEGMENTS WERE FOUND TO BE MISSING
```

Where the logical child segment being processed is a bidirectional logical child, the message below is produced. It states the total number of paired logical child segment occurrences found to have been physically deleted.

```
.......NO....... PAIRED LOGICAL CHILD SEGMENTS WERE FOUND TO BE MISSING
```

The DAZELORE Turbo procedure reports additionally the highest DP ISN found and how many DP ISNs are bigger than MAXDPISN.

The following message is produced last.

```
LOGICAL RELATIONSHIP SUCCESSFULLY ESTABLISHED
```

Restart Considerations

The Establish Logical Relationship utility (DAZELORE) may be run with checkpoints in order to make it restartable by specifying "RESTART=YES" as DAZELORE parameter. The number of logical child segment occurrences to be processed before a checkpoint is taken are defined with the INTER parameter. When checkpoints are being used and a DAZELORE run is terminated abnormally, it may be restarted from any checkpoint which has been successfully issued. The procedure for taking checkpoints and restarting runs is the same as that used for all normal batch jobs issuing restart and symbolic checkpoint calls (see the section *Recovery and Restart Procedures* in the *ADL Interfaces* documentation for more details). For every checkpoint, an Adabas unsynchronized checkpoint (a C1 call) is taken and a message is written to DAZOUT1 naming the checkpoint ID. An abnormally terminated DAZELORE run may be restarted after the Adabas file(s) involved have been restored to the situation as represented by the checkpoint from which it is to be restarted. DAZELORE will reposition itself automatically when restarted from a checkpoint.

A restart is not possible in case of running DAZELORE with MODE=STANDARD in a z/VSE system.

For extremely long DAZELORE runs, it is possible to limit the total number of logical child segment occurrences to be processed in one particular run and to use the restart capabilities to continue processing in a subsequent run. For this limitation "RESTART=YES" must be used and both the INTER and the NUMCP parameters must be specified. As described above, the INTER parameter specifies the number of logical child segment occurrences to be processed before a checkpoint is taken whereas the NUMCP parameter specifies the number of checkpoints to be written before the program ends. The total number of processed logical child segment occurrences is therefore the product of both parameters: NUMCP * INTER. When the checkpoints specified with the NUMCP parameter are written, DAZELORE ends. This termination of DAZELORE does not delete the checkpoint entries in the ADL directory file. This means that a subsequent DAZELORE run may be restarted from the last checkpoint without the Adabas files involved being restored, as no Adabas calls were issued after the last checkpoint. For the restart use "CPID=*cccc*" as DAZIFP parameter where cccc is the last checkpoint Id.

When all logical child occurrences have been processed, DAZELORE ends normally and deletes all checkpoints from the ADL directory.

The additional keyword parameters for DAZELORE may be specified in the parameter statement as follows:

```
LC=LCname,MODE=mode,keyword
```

Parameters for DAZELORE

Keyword	Explanation	Possible values:	Default:
RESTART	Specifies whether DAZELORE can be restarted or not. If "YES" is specified, DAZELORE can be restarted. The parameters INTER and NUMCP can be specified. DAZELORE issues XRST and CHKP calls. The ADL Directory file is used in exclusive mode, i.e. no other user can work on it simultaneously. If "NO" is specified, DAZELORE cannot be restarted. In case of an unexpected failure, the utility must be started from the very beginning. The ADL Directory file is not used in exclusive mode.	YES NO	YES
INTER	Specifies the number of processed logical child segment occurrences before a checkpoint is issued. The parameter can only be specified for RESTART=YES. For MODE=TURBO, checkpoints are issued after all logical child segments belonging to one parent segment have been processed. That means that the real number of processed logical child segment occurrences can be slightly higher than specified with the `INTER` parameter.	1 - 2147483647	2147483647
NUMCP	Specifies the number of checkpoints issued by DAZELORE. As soon as DAZELORE has reached this number, it is stopped and can be restarted later. At the restart the last checkpoint Id must be given as DAZIFP CPID parameter. The `NUMCP` parameter can only be specified for RESTART=YES and if the `INTER` parameter has been specified. The total number of processed logical child segment occurrences is: `NUMCP` * `INTER`.	1 - 2147483647	1
MAXDPISN	Specifies the length (in bytes) of the table of the destination parent (DP) counters (the destination parent is the logical parent in the logical relationship). As far as possible the table should be as big as the maximum ISN of the destination parent data. The counters of DPs with ISNs higher than MAXDPISN are updated immediately and lead therefore to a poorer performance. The report of DAZELORE outlines the highest DP ISN found and how many DP ISNs are bigger than MAXDPISN. The REGION/PARTITION of the job should be adjusted accordingly so that the table can be allocated. This parameter can only be specified for MODE=TURBO.	1 - 16777215	65536

Example

Where the total number of logical child segment occurrences is 5,312,726, and it has been decided to process them in 3 runs, with a checkpoint to be taken every 10,000th logical child segment occurrence, the following parameter setting is required:

run	INTER	NUMCP	Processed LCs
1	10,000	200	2,000,000
2	10,000	200	2,000,000
3	10,000	200	1,312,726

z/OS JCL Requirements

The following examples illustrate the z/OS JCL requirements for the utility runs described in this section. They include the requirements for the `DAZUNDLI`, `DAZUNLOD` and `DAZREFOR` utilities, for the `ADACMP` step during data loading, and for `DAZELORE`.

DAZUNDLI

The table below lists the data sets used by the Unload utility, `DAZUNDLI`.

DDname	Medium	Description
DAZIN1	Reader	Control input for DAZUNDLI. *
DAZIN2	Reader	Control input for batch monitor in mixed mode.
DAZIN3	Tape/Disk	Root sequence field values. **
DAZOUT1	Printer	Report, messages and codes.
DAZOUT3	Tape/Disk	The unloaded database.

* Set to dummy if no keyword is specified.

** Only required if the ROOTKEYS=SEQ is set.

Example

```
//UNDLI       EXEC DLIBATCH,PSB=PSB4CON,MBR=DAZIFP          (1)
//STEPLIB     DD
//            DD
//            DD   DSN=ADLvrs.LOAD,DISP=SHR
//            DD   DSN=ADABAS.LOAD,DISP=SHR
//*
//* DATASETS DESCRIBING DL/I DATABASES
//*
//G.file      DD   ...
```

```
//*
//* ADARUN CARDS
//*
//G.DDCARD   DD   *
ADARUN PROGRAM=USER,...
//*
//* ADABAS DL/I BRIDGE DATASETS
//*
//G.DAZIN1   DD *
MODE=CHECKNUM
//G.DAZIN2   DD   *
UNL,DAZUNDLI,PSB4CON
//G.DAZOUT1  DD   SYSOUT=X
//G.DAZOUT3  DD   DSN=ADL.DBD4.UNLOAD,DISP=OLD
```

(1) The standard batch procedure provided by IBM as part of the IMS/DB installation.

DAZUNLOD

The following table lists the data sets used by the Unload utility, `DAZUNLOD`.

DDname	Medium	Description
DAZOUT1	Printer	Report, messages and codes.
DAZOUT3	Tape/Disk	The unloaded database.

Example

```
//UNLOD      EXEC DLIBATCH,PSB=PSB4CON,MBR=DAZUNLOD        (1)
//G.STEPLIB  DD
//          DD
//          DD   DSN=ADLvrs.LOAD,DISP=SHR
//*
//* DATASETS DESCRIBING DL/I DATABASES
//*
//G.file     DD   ...
//*
//* ADABAS DL/I BRIDGE DATASETS
//*
//G.DAZOUT1  DD   SYSOUT=X
//G.DAZOUT3  DD   DSN=ADL.DBD4.REFOR,DISP=OLD
```

(1) The standard batch procedure provided by IBM as part of the IMS/DB installation.

DAZREFOR

The following table lists the data sets used by the Reformat utility, `DAZREFOR`.

DDname	Medium	Description
DAZIN3	Tape/Disk	The data to be reformatted.
DAZOUT1	Printer	Report, messages and codes.
DAZOUT3	Tape/Disk	The unloaded database.

Example

```
//REFOR        EXEC PGM=DAZIFP,PARM='REF,DAZREFOR,PSB4CON'
//STEPLIB      DD   DSN=ADLvrs.LOAD,DISP=SHR
//             DD   DSN=ADABAS.LOAD,DISP=SHR
//DAZIN3       DD   DSN=ADL.DBD4.REFOR,DISP=OLD
//DAZOUT1      DD   SYSOUT=X
//DAZOUT2      DD   SYSOUT=X
//DAZOUT3      DD   DSN=ADL.DBD4.UNLOAD,DISP=OLD
//DDCARD       DD   *
ADARUN PROGRAM=USER,...
```

Loading the Data

The table below lists the data sets used during the first step of the data loading process, which uses the Adabas Compression utility, `ADACMP`.

DDname	Medium	Description
DAZIN1	Reader	Control input for the Adabas User Exit 6, DAZUEX06.
DAZOUT1	Printer	Report, messages and codes.

Examples

1. *ADACMP run for an initial load of DBD4 on DBID=9,FNR=34*

```
//ADACMP   EXEC PGM=ADARUN
//STEPLIB  DD   DSN=ADABAS.LOAD,DISP=SHR
//         DD   DSN=ADLvrs.LOAD,DISP=SHR
//DDEBAND  DD   DSN=ADL.DBD4.UNLOAD,DISP=OLD
//DDAUSBA  DD   DSN=ADL.DBD4.LOAD,DISP=OLD
//DDFEHL   DD   DSN=&&FEHL,UNIT=SYSDA,SPACE=(CYL,(1,1)),DISP=(,PASS)
//DDDRUCK  DD   SYSOUT=X
//DDPRINT  DD   SYSOUT=X
//DDCARD   DD   *
ADARUN PROGRAM=ADACMP,DB=9,SVC=svc,DE=3390,UEX6=I00034
//DDKARTE  DD   *                            (1)
ADACMP USERISN
ADACMP FNDEF='01,Z0,A,DE,MU,UQ,NU'                      Z1 INVERTED (INSERT ↩
LAST)
ADACMP FNDEF='01,Z1,A,DE,MU,UQ,NU'                          MAIN DESCRIPTOR FIELD ↩
UP22
ADACMP FNDEF='01,Z2,004,B,NU'                             ROOT ISN (NON ROOT SEGS) ↩

.
.
/*
//DAZIN1   DD   *
FNR=034,MODE=LOAD
//DAZOUT1  DD   SYSOUT=X
//*
//         EXEC PGM=IEBGENER
//SYSPRINT DD   SYSOUT=X
//SYSUT1   DD   DSN=&&FEHL,DISP=(OLD,DELETE)
//SYSUT2   DD   SYSOUT=X
//SYSIN    DD   *
/*
```

(1) The ADACMP statements generated by the CBC utility as member W00034.

2. *ADACMP run for a mass update of the logical child D5B*

```
//ADACMP   EXEC PGM=ADARUN
//STEPLIB  DD   DSN=ADABAS.LOAD,DISP=SHR
//         DD   DSN=ADLvrs.LOAD,DISP=SHR
//DDEBAND  DD   DSN=ADL.DBD5.UNLOAD,DISP=OLD
//DDAUSBA  DD   DSN=ADL.DBD4.MASS,DISP=OLD
//DDFEHL   DD   DSN=&&FEHL,UNIT=SYSDA,SPACE=(CYL,(1,1)),DISP=(,PASS)
//DDDRUCK  DD   SYSOUT=X
//DDPRINT  DD   SYSOUT=X
//DDCARD   DD   *
ADARUN PROGRAM=ADACMP,DB=9,SVC=svc,DE=3390,UEX6=I00035
//DDKARTE  DD   *                          (1)
ADACMP FNDEF='01,Z0,A,DE,MU,UQ,NU'                          Z1 INVERTED (INSERT ↩
LAST)
ADACMP FNDEF='01,Z1,A,DE,MU,UQ,NU'                          MAIN DESCRIPTOR FIELD ↩
UP22
ADACMP FNDEF='01,Z2,004,B,NU'                             ROOT ISN (NON ROOT SEGS) ↩

.
.
/*
//DAZIN1   DD   *
FNR=034,LC=D5B,MODE=MASS
//DAZOUT1  DD   SYSOUT=X
//*
//         EXEC PGM=IEBGENER
//SYSPRINT DD   SYSOUT=X
//SYSUT1   DD   DSN=&&FEHL,DISP=(OLD,DELETE)
//SYSUT2   DD   SYSOUT=X
```

```
//SYSIN    DD   *
/*
```

(1) The `ADACMP` statements generated by the `CBC` utility as member `W00034` without the `USERISN` option.

DAZELORE

The table below lists the data sets used by the Establish Logical Relationship utility, `DAZELORE`.

DDname	Medium	Description
DAZIN1	Reader	Control input for DAZELORE and for the batch monitor in normal mode.
DAZIN2	Reader	Control input for the batch monitor in mixed mode. *
DAZOUT1	Printer	Report, messages and codes.

* Only required if `DAZELORE` is run in mixed mode, i.e. when `MODE=STANDARD` is specified for the conversion of an original DL/I database.

Examples

1. *DAZELORE run with MODE=STANDARD*

```
//ELORE      EXEC DLIBATCH,PSB=PSB4BI,MBR=DAZIFP    (1)
//G.STEPLIB  DD
//          DD
//          DD   DSN=ADLvrs.LOAD,DISP=SHR
//          DD   DSN=ADABAS.LOAD,DISP=SHR
//*
//* DATASETS DESCRIBING DL/I DATABASES DBD4 AND DBD5
//*
//G.FILE     DD   ...
//*
//* ADARUN CARDS
//*
//G.DDCARD   DD   *
ADARUN PROGRAM=USER,...
//*
//* ADABAS DL/I BRIDGE DATASETS
//*
//G.DAZIN1   DD   *
LC=D4F,MODE=STANDARD
//G.DAZIN2   DD   *
ELO,DAZELORE,PSB4BI
//G.DAZOUT1  DD   SYSOUT=X
```

(1) The standard batch procedure provided by IBM as part of the IMS/DB installation.

2. *DAZELORE run with MODE=TURBO*

```
//         EXEC PGM=DAZIFP,PARM='ELO,DAZELORE,INSTELO'
//STEPLIB   DD DSN=ADLvrs.LOAD,DISP=SHR
//          DD DSN=ADABAS.LOAD,DISP=SHR
//DAZOUT1   DD SYSOUT=X
//DDCARD    DD *
ADARUN PROGRAM=USER,...
//DAZIN1    DD *
LC=COURSEP,MODE=TURBO,RESTART=NO,MAXDPISN=5000000
```

z/VSE JCS Requirements

The following examples illustrate the JCS requirements for the utility runs described in this section. They include the requirements for the DAZUNDLI, DAZUNLOD and DAZREFOR utilities, for the ADACMP step during data loading, and for DAZELORE.

- DAZUNDLI
- DAZUNLOD
- DAZREFOR
- Loading the Data
- DAZELORE

DAZUNDLI

The following table lists the files used by the Unload utility, DAZUNDLI, in mixed mode.

DTF	Logical Unit	Medium	Description
DAZIN1	SYSIPT	Reader	Control input for DAZUNDLI. *
DAZIN2	SYSIPT	Reader	Control input for batch monitor.
DAZIN3D	SYS014	Disk	Root seq. field value **/***
DAZIN3T	SYS014	Tape	Root seq. field value **/***
DAZOUT1	SYSLST	Printer	Report, messages and codes.
DAZOT3D	SYS013	Disk	The unloaded database. **
DAZOT3T	SYS013	Tape	The unloaded database. **

* Only required if any keyword has been specified.

** Only one of either disk or tape is required. The logical unit indicated is the default logical unit. To change it, specify the SQ parameter either in the ADL parameter module or as a dynamic parameter, for example: SQ=(5,)

*** Only required if the ROOTKEYS=SEQ keyword has been specified.

The control input for the batch monitor (`DAZIFP`), for `ADARUN`, for `DAZUNDLI`, and for the DL/I initialization program, `DLZRRC00`, are all read from `SYSIPT`. The control statements must be specified in the following order:

```
DLI,DAZIFP,psbname,...                      input for DLZRRC00
UNL,DAZUNDLI,psbname,...                    input for DAZIFP
ADARUN DB=dbid,MO=MULTI,PROGRAM=USER,...    input for ADARUN
/*
MODE=CHECKNUM                               input for DAZUNDLI
```

Examples

1. *DAZUNDLI run unloading a database on disk*

   ```
   // ASSGN SYS013,DISK,VOL=volser,SHR
   // DLBL DAZOT3D,'ADL.DBD4.UNLOAD,0,SD'
   // EXTENT SYS013,volser,1,0,.......
   // EXEC DLZRRC00
   DLI,DAZIFP,PSB4CON
   UNL,DAZUNDLI,PSB4CON
   ADARUN PROGRAM=USER,...
   /*
   ```

2. *DAZUNDLI run unloading a database on tape*

   ```
   // ASSGN SYS005,TAPE
   // TLBL DAZOT3T,'ADL.DBD4.UNLOAD,0,SD'
   // EXEC DLZRRC00
   DLI,DAZIFP,PSB4CON
   UNL,DAZUNDLI,PSB4CON,SQ=(5,)                                              ↩

   ADARUN PROGRAM=USER,...
   /*
   ```

DAZUNLOD

The table below lists the files used by the Unload utility, `DAZUNLOD`.

DTF	Logical Unit	Medium	Description
DAZOUT1	SYSLST	Printer	Report, messages and codes.
DAZOT3D	SYS014	Disk	The unloaded database. *
DAZOT3T	SYS014	Tape	The unloaded database. *

* Only one of the two is required. The logical unit indicated is the default logical unit. To change it, specify the `SQ` parameter either in the ADL parameter module or as a dynamic parameter, for example: `SQ=(5,)`

Example

```
// ASSGN SYS013,DISK,VOL=volser,SHR
// DLBL DAZOT3D,'ADL.DBD4.REFOR,0,SD'
// EXTENT SYS013,volser,1,0,.......
// EXEC DLZRRC00
DLI,DAZUNLOD,PSB4CON
/*
```

DAZREFOR

The following table lists the files used by the Reformat utility, `DAZREFOR`.

DTF	Logical Unit	Medium	Description
DAZIN1	SYSIPT	Reader	Control input for batch monitor.
DAZOUT1	SYSLST	Printer	Report, messages and codes.
DAZIN3D	SYS014	Disk	The unloaded database. *
DAZIN3T	SYS014	Tape	The unloaded database. *
DAZOT3D	SYS013	Disk	The unloaded database. *
DAZOT3T	SYS013	Tape	The unloaded database. *

* Only one of either tape or disk is required. The logical unit indicated is the default logical unit. To change it, specify the `SQ` parameter either in the ADL parameter module or as a dynamic parameter, for example: `SQ=(5,)`

The control input for the batch monitor (`DAZIFP`) and for `ADARUN` is read from `SYSIPT`. The control statements must be specified in the following order:

```
REF,DAZREFOR,psbname,...                                               input for ↩
DAZIFP
ADARUN DB=dbid,MO=MULTI,PROGRAM=USER,...    input for ADARUN
/*
```

Example

```
// ASSGN SYS014,DISK,VOL=volser,SHR
// DLBL DAZIN3D,'ADL.DBD4.REFOR,0,SD'
// EXTENT SYS014,volser,1,0,.......
// ASSGN SYS013,DISK,VOL=volser,SHR
// DLBL DAZOT3D,'ADL.DBD4.UNLOAD,0,SD'
// EXTENT SYS013,volser,1,0,.......
// EXEC DAZIFP
REF,DAZREFOR,PSB4CON
ADARUN PROGRAM=USER,...
/*
```

Loading the Data

The table below lists the files used during the first step of the data loading process, which uses the Adabas Compression utility, `ADACMP`.

DTF	Logical Unit	Medium	Description
DAZIN1	SYSIPT	Reader	Control input for the Adabas User Exit 6, DAZUEX06.
DAZOUT1	SYSLST	Printer	Report, messages and codes.

The control input for the Adabas User Exit 6, `DAZUEX06`, for `ADARUN` and for the Adabas Compression utility, `ADACMP`, are all read from `SYSIPT`. The control statements must be specified in the following order:

```
ADARUN DB=dbid,MO=MULTI,PROGRAM=ADACMP    input for ADARUN
/*
ADACMP USERISN,RECFM=VB,LRECL=8196        input for ADACMP
ADACMP FNDEF='01,Z0,.....'                                          ↩

.
.
/*
FNR=fnr,MODE=LOAD                         input for DAZUEX06
/*
```

Examples

1. *ADACMP run for an initial load of DBD4, DBID=9,FNR=34*

```
// ASSGN SYS010,DISK,VOL=volser,SHR
// DLBL EBAND,'ADL.DBD4.UNLOAD,0,SD'
// EXTENT SYS010,volser,1,0,.......
// ASSGN SYS012,DISK,VOL=volser,SHR
// DLBL AUSBA,'ADL.DBD4.LOAD,0,SD'
// EXTENT SYS012,volser,1,0,.......
// ASSGN SYS014,IGN
// EXEC PROC=ADLLIBS
// EXEC ADARUN,SIZE=4K
ADARUN PROGRAM=ADACMP,SVC=svc,DE=3390,UEX6=I00034
/*
ADACMP USERISN,LRECL=8196,RECFM=VB          (1)
ADACMP FNDEF='01,Z0,A,DE,MU,UQ,NU'                  Z1 INVERTED (INSERT ↩
LAST)
ADACMP FNDEF='01,Z1,A,DE,MU,UQ,NU'                 MAIN DESCRIPTOR FIELD  ↩

ADACMP FNDEF='01,Z2,003,B,NU'                              ROOT ISN (NON ↩
ROOT SEGS)
.
.
/*
```

```
FNR=034,MODE=LOAD
/*
```

(1) The `ADACMP` statements generated by the `CBC` utility as member `W00034`.

2. *ADACMP run for a mass update of the logical child D5B*

```
// ASSGN SYS010,DISK,VOL=volser,SHR
// DLBL EBAND,'ADL.DBD5.UNLOAD,0,SD'
// EXTENT SYS010,volser,1,0,.......
// ASSGN SYS012,DISK,VOL=volser,SHR
// DLBL AUSBA,'ADL.DBD4.MASS,0,SD'
// EXTENT SYS012,volser,1,0,.......
// ASSGN SYS014,IGN
// EXEC PROC=ADLLIBS
// EXEC ADARUN,SIZE=4K
ADARUN PROGRAM=ADACMP,SVC=svc,DE=3390,UEX6=I00034
/*
ADACMP LRECL=8196,RECFM=VB (1)
ADACMP FNDEF='01,Z0,A,DE,MU,UQ,NU'      Z1 INVERTED (INSERT LAST)
ADACMP FNDEF='01,Z1,A,DE,MU,UQ,NU'      MAIN DESCRIPTOR FIELD
ADACMP FNDEF='01,Z2,003,B,NU ROOT'      ISN (NON ROOT SEGS)
.
.
/*
FNR=34,MODE=MASS,LC=D5B
/*
```

(1) The `ADACMP` statements generated by the `CBC` utility as member `W00034` without the `USERISN` option.

DAZELORE

The following table lists the files used by the Establish Logical Relationship utility, `DAZELORE`.

DTF	Logical Unit	Medium	Description
DAZIN1	SYSIPT	Reader	Control input for DAZELORE and for the batch monitor in normal mode.
DAZIN2	SYSIPT	Reader	Control input for the batch monitor in mixed mode. *
DAZOUT1	SYSLST	Printer	Report, messages and codes.

* Only required if `DAZELORE` is run in mixed mode, i.e. when `MODE=STANDARD` is specified for the conversion of an original DL/I database.

The control input for the batch monitor, `DAZIFP`, for `DAZELORE` itself, for `ADARUN` and for the DL/I initialization program, `DLZRRC00`, is read from `SYSIPT`. The control statements must be specified in the following order:

```
DLI,DAZIFP,psbname,...                      input for DLZRRC00 *
ELO,DAZELORE,psbname,...                    input for batch monitor
/*
ADARUN DB=dbid,MO=MULTI,PROGRAM=USER,...    input for ADARUN
/*
LC=lcname,MODE=STANDARD                     input for DAZELORE
/*
```

* Only required when DAZELORE is run in mixed mode.

Examples

1. *DAZELORE run with MODE=STANDARD*

```
// EXEC DLZRRC00
DLI,DAZIFP,PSB4CON
ELO,DAZELORE,PSB4CON
/*
ADARUN PROGRAM=USER,...
/*
LC=D4F,MODE=STANDARD
/*
```

2. *DAZELORE run with MODE=TURBO*

```
// EXEC DAZIFP
ELO,DAZELORE,INSTELO
/*
ADARUN PROGRAM=USER,...
/*
LC=COURSEP,MODE=TURBO,RESTART=NO,MAXDPISN=5000000                      ↩

/*
```

7 Migration of a GSAM Data Base

This chapter covers the following topics:

Introduction

A GSAM database uses the Generalized Sequential Access Method (`GSAM`) under DL/I. It is recognized by the `'ACCESS=GSAM'` keyword for the DBD statement in the DBD definition. A `GSAM DBD` does not specify any segment or field definitions. The corresponding PCB statement in the PSB definition uses `'TYPE=GSAM'`.

For more information regarding the definition of a `GSAM DBD` refer to the IMS Utilities Reference Manual of IBM.

Restrictions

`GSAM` databases are supported by ADL with some restrictions regarding the DBD definition, the JCL and the `OPEN` command.

In particular, the following restrictions are valid for a `GSAM DBD`:

- The records must have fixed length, i.e. the `RECFM` keyword of the `DATASET` statement in the DBD definition must have a value of `'F'` or `'FB'`.
- The record length (`RECORD` keyword) must be at least 12 bytes.

If these conditions are met, the `GSAM DBD` can be converted and the data can be loaded into Adabas, as described in the next section.

Like for any other converted database, the data set corresponding to a converted GSAM database cannot longer be referenced in the JCL, because the data is in Adabas. Consequently, the record format (`RECFM`) parameter cannot be overwritten in the JCL. With DL/I the disposition (`DISP`) of the output dataset defines, whether a new dataset is created or whether the data is added to the end of the dataset. With ADL the data is always added to the end. If you want to insert data from the beginning of a data set, which does already contain data, you must refresh the Adabas file before your program starts. This can be performed by either using the `REFRESH` function of the Adabas `ADADBS` utility or manually with the Adabas Online System.

The DL/I `OPEN` and `CLSE` calls are ignored by ADL. Under DL/I you can add an option to the `OPEN` call, which specifies the kind of control character in the first byte of each record in the output data set. With ADL there is always no control character in the first byte of each output record.

With DL/I an application can use the `CLSE` and `OPEN` commands to read a `GSAM` database after it was loaded, without terminating in between. This works in the same way after the migration, because ADL does not need these calls for a re-positioning.

Conversion of a GSAM Data Base

Run the `GSAM DBD` through the step 1 (assembly), step 2 (CBC utility) and step 3 (create input decks) of the ADL control block conversion (CBC). For a more detailed description of the ADL CBC, see the section *ADL Conversion Utilities for DBDs and PSBs* in this documentation. Step 4 (assemble and link-edit Adabas user 6 extension) is not required for a `GSAM` database.

ADL adds a segment `'GSAMROOT'` and a sequence field `'GSAMFLD'` to the DBD definition. The length of the segment is the same as specified with the `'RECORD'` keyword in the `DATASET` statement. The field always has a length of 12 bytes. Additionally, ADL adds a sensitive segment `'GSAMROOT'` to the `GSAM PCB`.

The `ADACMP` cards which are generated by step 3 of the ADL conversion utility is used as input for the `ADACMP COMPRESS` utility. Note, that the `USERISN=YES` option must not be used for any Adabas utility run against a file of a converted `GSAM DBD`.

There is no need to run the ADL unload utility when unloading the DL/I `GSAM` database. Instead, the sequential file can directly be used as input for the Adabas compression utility, by specifying the `GSAM` dataset as `DDEBAND` for `ADACMP`. You must not use the ADL user exit 06 for the compression.

Use the following parameter settings for `ADALOD` when loading the data into Adabas:

```
DATAPFAC=1
DSREUSE=NO
ISNREUSE=NO
USERISN=NO
```

Features of a Converted GSAM Data Base

The `GN`, `ISRT` and `GU` calls are supported in the same way as under DL/I. `GN` and `ISRT` can be specified with or without a record search argument (`RSA`). With ADL the first 4 bytes of the `RSA` contain the Adabas ISN of the record and the last 4 bytes contain hexadecimal zeros. After a successful call, the `RSA` is also placed into the Key Feedback Area of the corresponding PCB. Further to these database calls, the system service calls `CHKP` and `XRST` are supported.

With the `LOAD` parameter of ADL, you can specify where the data of an `ISRT` call against a PCB with `PROCOPT=L` or `LS` is to be written. See the section *ADL Parameter Module* in the *ADL Installations* documentation for more information about the `LOAD` parameter. When you specify `'LOAD=UTILITY'` the data is written to a variable blocked sequential file. For a `GSAM` database, this file can be loaded back into Adabas in the same way as described above. Especially you must set `USERISN=NO`, and you must not use the ADL user exit 6.

When '`LOAD=DIRECT`' is specified as ADL parameter, you can insert and read data simultaneously in one application by using different PCBs in the PSB. In particular you do not need to issue a `CLSE` and an `OPEN` call, when you want to retrieve data after inserting it. You can also mix the `ISRT` and `GET` calls as you like, and the position is kept for each PCB.

Further to inserting and retrieving data, you can also replace and delete records, by using the `REPL` (replace) and `DLET` (delete) calls. The corresponding PCB must specify `PROCOPT=R` or `=D`, respectively. The data storage of a deleted record is not released, so that a new inserted record is put to the end of the database, as expected for `GSAM` databases.

Note: An Adabas backout after a delete request acts like a re-insert, i. e. the record is inserted to the end of the database. After such a delete/backout the data will be therefore in another sequence than before.

After the data conversion you can manipulate it with Natural. This includes update and delete requests. Because there is no ADL internal pointer field stored with the data, there is no need to use the ADL Consistency.

ADL translates a get request against a `GSAM DBD` to an Adabas `L2` (read physical) call. For a high performance it is recommended to use the Adabas Multifetch feature for a `GN` sequence against a converted `GSAM` database. Refer to *Using the Adabas Multifetch Feature*, section *Performance Considerations* in the *ADL Interfaces* documentation for more information about the usage of the Adabas Multifetch feature against migrated data. In particular, you should make sure, that the correct Adabas command ID is prefetched.

Index