Adabas for Linux, UNIX and Windows

Adabas Basics

Version 6.7

October 2018

ADABAS & NATURAL

This document applies to Adabas for Linux, UNIX and Windows Version 6.7 and all subsequent releases.

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

Document ID: ADAOS-BASICS-67-20211006

Table of Contents

Adabas Basics

This document contains basic information about Adabas databases. This includes aspects of database design, the components that make up an Adabas database, the field definition table and the available definition options, loading data into a database and unloading data from a database, as well as information about using the Adabas utilities.

The following topics are covered:

- ***Database Design*** contains information on database design. It includes information on Adabas file structures, multiple-value fields and periodic groups, record design, the use of keys (descriptors), disk space usage (compression, null value suppression, padding factors), security planning and restart and recovery planning.
- ***Container Files*** provides information about the Adabas container files.
- ***Temporary Working Space*** contains information about allocating and deleting temporary working space if it is required by the Adabas nucleus or utilities.
- ***FDT Record Structure*** describes how to define the record structure of a file in the database.
- ***Defining Descriptors*** defines how to create and delete descriptors for a file.
- ***Loading and Unloading Data*** describes various methods for loading data into the database and unloading data from the database.
- ***Using Utilities*** provides general information on how to work with the Adabas utilities.
- ***User Exits and Hyperexits*** contains an explanation of the user exits and hyperexits that are supported by Adabas.
- ***Adabas on Read-only Devices*** contains information about running Adabas on read-only devices, and any restrictions that apply in such an environment.

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at **https://documentation.softwareag.com**.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at **https://empower.softwareag.com/**.

You can find product information on the Software AG Empower Product Support website at **https://empower.softwareag.com**.

To submit feature/enhancement requests, get information about product availability, and download products, go to **Products**.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the **Knowledge Center**.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at **https://empower.softwareag.com/public_directory.aspx** and give us a call.

Software AG Tech Community

You can find documentation and other technical information on the Software AG Tech Community website at **https://techcommunity.softwareag.com**. You can:

- Access product documentation, if you have Tech Community credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

2 Database Design

Technical Introduction to Adabas

If a program is to access data in an Adabas database, it must issue Adabas commands (for further information, refer to the *Command Reference* section). Because the Adabas direct-call command interface is a low-level interface, Software AG also offers several higher-level interfaces to Adabas:

- The development environment Natural, where the access to Adabas is integrated in the programming language.
- The Adabas SQL Gateway, which is an SQL interface for Adabas.
- The Adabas SOA Gateway, which is the Adabas interface into a service-oriented architecture (SOA).

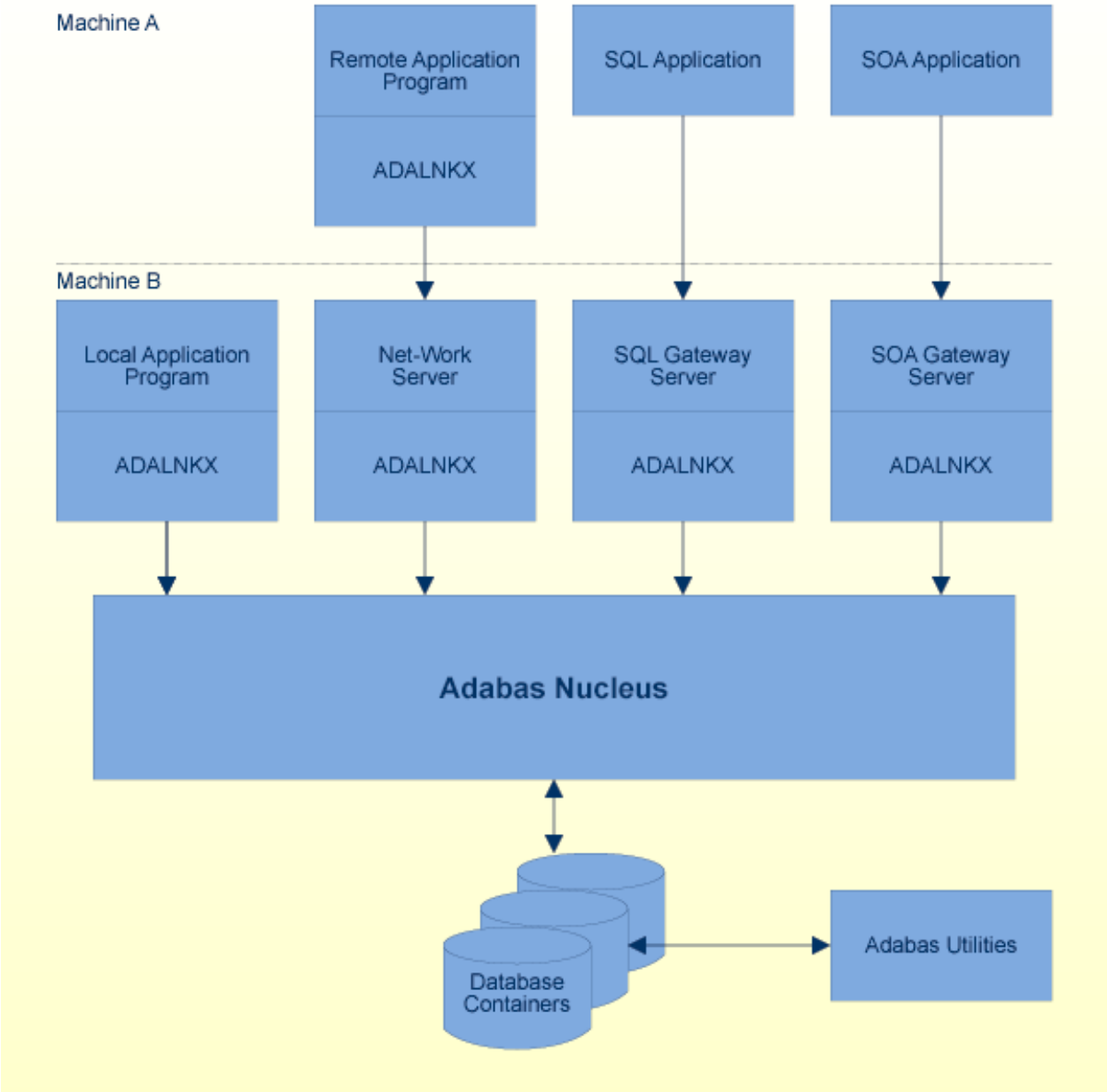
A program that issues Adabas commands is called an Adabas client. The Adabas commands are executed by a database server called the Adabas nucleus.

In order to access the database, the Adabas client must be linked with an Adabas interface, for example ADALNKX, which is part of the Adabas client package. For further information, refer to *Command Reference*, *Linking Application Programs*.

The Adabas client may either run locally on the same (physical or virtual) machine as the Adabas nucleus, or remotely on another machine. The additional product Entire Net-Work is required for remote access.

In addition to the Adabas nucleus, there are also a number of Adabas utilities for database administration purposes, which access the Adabas database.

The following figure shows how programs access Adabas:



Note: The database containers can also be stored on separate storage servers.

Caution: The database containers may be accessible from different machines. However, *the consistency of the database is only guaranteed if all utility and nucleus processes accessing the database containers of a database concurrently are running on the same machine*, because IPC resources only visible on the same machine are used for the synchronization between the utility and nucleus processes. An exception is remote utilities that do not access the database containers, but perform special remote Adabas calls. When all utility and nucleus processes for a database have been terminated, you can restart the database processes on another

machine, but the database administrator is responsible to ensure that no database process is started on the new machine as long as a database process for the database is still active on the old machine; this is not checked by Adabas.

Performance Control During System Design

The performance of a system is measured by the time and computer resources required to run it. These may be important for the following reasons:

- Some system functions may have to be completed within a specified time;
- The system may compete for computer resources with other systems which have more stringent time constraints.

Performance may not, however, be the most important objective. Trade-offs will often have to be made between performance and the following:

- Flexibility;
- Data independence;
- Accessibility of information;
- Security considerations;
- Currency of information;
- Ease of scheduling and impact on concurrent users of the database;
- Disk space.

In some cases, performance may be a constraint to be met rather than an objective to be optimized. If the system meets its time and volume requirements, attention may be turned from performance to other areas.

Methodology for Performance Control in System Design

The need to achieve satisfactory performance may affect one or more of the following:

- Hardware design;
- The design of the database;
- The options used when loading data into the database;
- The logic of application functions (for example, whether to use direct access or a combination of sorts and sequential accesses);
- Operations procedures and scheduling.

Performance requirements must be considered early in the system design process. The following procedure may be used as a basis for controlling performance:

1. Obtain from the users the time constraints for each major system function. These requirements are likely to be absolute, i.e., the system is probably useless if it cannot meet them;
2. Evaluate if the available hardware resources are sufficient, either by experience with other databases, or by simulating the expected load. If you need new hardware consider the following:
   - You may have to decide between computers with a small number of fast, but expensive hardware threads, and computers with a large number of cheap, but relative slow hardware threads. Often the computers with the large number of hardware threads provide larger computing power for less money, but only if the parallelism can really be used. Adabas is able to use these hardware threads in parallel, when there are enough parallel users, but if you use Adabas mainly to run sequential batch programs without parallelism, Adabas is not able to work in parallel.
   - The hardware threads may be distributed across more than one socket, and the sockets may be distributed across more than one board. In such a case, the synchronization times between the hardware threads may be significantly longer when they are on different sockets than when they are on the same socket. A synchronization is always required when a database block is read from the buffer pool, and because each database command must read several database blocks (index and data blocks, address converter, file control block, field description table), the advantage of having more computing power by means of additional threads may be outranged by the increased synchronization times if additional threads are on a different socket or board.

     Another problem is that the memory is usually distributed across the sockets; access to the socket's own memory is relatively fast, while access to the memory belonging to a different socket is relatively slow, especially if it is located on a different board. Depending on the hardware, the access times to remote and local memory differ by a factor of a little more than 1 and about 3 (NUMA architecture: Non-Uniform Memory Access).

     Therefore, you should restrict ADANUC to hardware threads on the same socket or board. This can be achieved with special operating system commands, or by using operating system virtualization concepts. Usually you get a better performance this way - the reduced synchronization costs and the faster memory access to local memory is larger than the advantage of more parallelization.

     In order to increase the performance on such modern architectures, Adabas uses the concept of Adabas Processing Units (ADANUC parameter APU): one Adabas processing unit (APU) consists of a command queue and its own threads where the commands are processed. If the operating system recognizes that the threads belonging to one APU belong together, and schedules them on the same socket/board, you may get a better performance than with binding the complete nucleus process to one socket/ board. Using the APU concept may also increase the performance, if the complete nucleus process is running on one socket, because the usage of more than one command queue reduces the number of lock conflicts compared to the usage of one large command queue.

However, Software AG cannot give a general recommendation for the optimal configuration of CPU binding and APU usage because the differences between the different operating systems and the different hardware implementations are simply too big. It may also be that APU usage brings no advantage at all. We recommend trying different configurations in your hardware environment and chosing the one that gives the best results for your database load.

- Consider usage of modern storage systems. They can avoid downtimes because of disk failures and provide good performance.
- Consider also the section ***Database Design, Recovery/Restart Design, Locations of Database Conatiners, Backup Files, and Protection Logs***.
- Adabas containers can be included in high availability clusters, but they are not directly supported by Adabas - it is necessary that you write the required scripts yourself. Load balancing in the clusters is not supported.

3. Describe each function in terms of the logical design model specifying:
   - The manner in which each record type is processed;
   - The access path and the sequence in which records are required;
   - The frequency and volume of the run;
   - The time available;
4. Decide which programs are most performance-critical. The choice may involve volumes, frequency, deadlines and the effect on the performance or scheduling of other systems. Other programs may also have minimum performance requirements which may constrain the extent to which critical functions can be optimized;
5. Optimize the performance of each critical function by shortening its access paths, optimizing its logic, eliminating database features which increase overheads, etc. In the first pass, an attempt should be made to optimize performance without sacrificing flexibility, accessibility of information, or other functional requirements of the system;
6. Estimate the performance of each critical function. If this does not yield a satisfactory solution, a relaxation of the time constraints or the functional requirements will have to be negotiated or a hardware upgrade may be required;
7. Estimate the performance of other system functions. Calculate the total cost and compare the cost and peak period resource requirements with the economic constraints. If the estimates do not meet the constraints, then a solution must be negotiated with the user, operations or senior management;
8. If possible, validate the estimates by loading a test database in order to time various functions. The test database should be similar to the planned one in terms of the number of records contained in each file and the number of values for descriptors. The size of each record is less important except for tests of sequential processing and then only if records are to be processed in something close to their physical sequence.

Unicode Support

Adabas supports Unicode on the basis of International Components for Unicode (ICU) libraries (V3.2). Please refer to the ICU homepage at **https://www.ibm.com/software/globalization/icu** for further information about ICU. The wide character field format (W) has been introduced for Unicode fields. The Adabas user can specify the external encoding used in Adabas calls or for the compression and decompression utilities ADACMP and ADADCU, but internally all data is stored in UTF-8.

The external encoding can be specified in:

- the Adabas OP command, where you can specify the default encoding for an Adabas session;
- the format and search buffers, where you can specify encodings at the field level;
- the utilities ADACMP and ADADCU, where you can specify the default encoding to be used during the execution of the utility.

  For search and sort operations, the Unicode byte order is not usually of much importance, but language-specific collations are - for this reason, Adabas supports collation descriptors. A collation descriptor generates a binary string from the original character string by applying Unicode Collation Algorithms and language-specific rules.

Mainframe Compatibility Considerations

The following points should be taken into consideration if you intend to write applications using Unicode character sets and when you intend to run the applications on both mainframes and UNIX/Windows platforms:

- Software AG recommends that you use the W format for text fields. The A format should only be used if the values only contain characters that are available in ASCII and EBCDIC, if the different ASCII and EBCDIC collations are not relevant, and if no collation descriptors are required.
- On mainframes, you can select the internal encoding yourself, but on UNIX/Windows platforms all W format fields are stored internally as UTF-8. Since the length of a string depends on its decoding, you should either use UTF-8 for internal encoding on mainframes as well, or you should ensure that your applications only store values that cannot overflow with the internal encoding on other platforms.
- On mainframes, W format fields are based on ECS, but on UNIX/Windows platforms they are based on ICU - you should, therefore, only use encodings that are available on all platforms.
- On UNIX/Windows platforms, collation descriptors are based on ICU, but on mainframes, the user has to provide user exits to generate collation descriptors. These user exits must generate collation sequences that are compatible with the ICU collations used on UNIX/Windows plat-

forms. On UNIX/Windows platforms, you must use the HE option in order to achieve the same behaviour as on mainframes.

- The Adabas direct commands on mainframes and UNIX/Windows platforms are not fully compatible with respect to their handling of W format fields. You should ensure that only compatible commands are used in cross-platform applications.

File and Record Design

It is possible to design an Adabas database with one file for each record type as identified during the conceptual design stage. Although such a structure would support any application functions required of it and is the easiest to manipulate in ad hoc queries, it may not be the best from the performance point of view:

- The number of Adabas calls would be increased. Each Adabas call requires interpretation, validation and queueing overhead;
- Accessing at least one index, Address Converter and Data Storage block from each of the files. In addition to the I/Os necessary for this process, it will require buffer pool space and perhaps result in the overwriting of blocks needed for a later request.

It is, therefore, often advisable to reduce the number of Adabas files used by critical programs. The following techniques may be used for this procedure:

- Using multiple-value fields and periodic groups;
- Using multiple record types in one Adabas file;
- Controlled data duplication.

Each of these techniques is described in the following sections.

Multiple-Value Fields and Periodic Groups

In the example shown below, ORDER-ITEM is defined as a periodic group in the ORDER file. Each order record contains a variable number of order items.

```
Order   Order   Date     Customer Item
Number  Date    Required Number   Code     Quantity

A1234E  29MAR   10JUN    UK432M   24801K   200     1ST OCCUR.
                                  30419T   100     2ND OCCUR.
                                  273952   300     3RD OCCUR.
```

A multiple-value field or a periodic group may be retrieved/updated in the same call and with the same I/Os as the main record. This can result in a saving in both CPU time and I/O requirements.

There are certain constraints concerning the usage of multiple-value fields and periodic groups that the user should be aware of:

- A periodic group cannot contain another periodic group;
- Depending on the compressed size of one occurrence, their usage can result in extremely large record sizes which may be larger than the maximum record size supported by Adabas.

Descriptors contained within a periodic group or derived from fields within a periodic group cannot be used as a sort key in FIND and SORT commands. In addition, specific rules apply to the methods in which search requests that involve one or more descriptors defined as multiple-value fields and/or contained within a periodic group may be used. These rules are described in the Command Reference Manual.

Multiple Record Types in a Single Adabas File

Another method of reducing the number of files is to store data belonging to two logical record types in the same Adabas file. For example, the figure Multiple Record Types (i) below shows how a customer file and an order file might be combined. This technique takes advantage of the Adabas null-value suppression facility.

Fields in the Field Definition Table for the combined file:

```
Key, Record Type, Order Data, Order Item Data
```

Stored records:

```
Key  Type  Order Data   *
```

```
Key  Type  * Order Item Data
```

* indicates suppressed null values.

The key of an order item record could be order number plus line sequence number within this order.

This technique reduces I/Os by allowing the customer and order record types to share various control blocks and higher-level index blocks. Thus fewer blocks have to be read before processing of the file can start, and more space is left free in the buffer pool for other types of blocks.

The customer and order records can be grouped together in Data Storage reducing the number of blocks which have to be read to retrieve all the orders for a given customer. If all the orders are added at the same time as the customer, the total I/Os required will also be reduced.

The key must be designed carefully to ensure that both customer and order data can be accessed efficiently. The key for a customer record will usually have the null value of the suffix which distinguishes different orders belonging to the same customer appended to it as shown in the figure Multiple Record Types (ii) below.

```
A00231    000               Order header for order A00231
A00231    001               Order item 1
A00231    002               Order item 2
A00231    003               Order item 3
A00232    000               Order header for order A00232
A00232    001               Order item 1
```

A record type field is not necessary if the program can tell whether it is dealing with a customer or order record by the contents of the key suffix. It may be necessary for a program to reread a record in order to read additional fields or else have Adabas return all the fields relevant in any of the record types.

Data Duplication

Physical Duplication

In some cases, a few fields from a header record are required almost every time a detail record is accessed. For example, the production of an invoice may require both the order item data and the product description which is part of the PRODUCT record. The simplest way to make this information quickly available to the invoicing program is to hold a copy of the product description in the order item data. This is termed physical duplication because it involves holding a duplicate copy of data which is already physically represented elsewhere.

Physical duplication can also be in effect if some fields from each detail record are stored as a periodic group in a header record.

Logical Duplication

Assume that a credit control routine needs the sum of all invoices sent to the customer. This information can be derived by reading and totalling the relevant invoices, but this might involve accessing randomly quite a large number of records. It can be obtained much more quickly if it is stored permanently in the customer record, provided it is correctly maintained. This is termed logical duplication because the duplicate information is not already stored elsewhere but is implied by the contents of other records.

Programs which update information that is physically or logically duplicated are likely to run more slowly because they must also update the duplicate copies. Physical duplication seldom causes much of a problem because it usually involves fields which are infrequently updated. Logical duplication almost always requires duplicate updating because the change of any one record can affect data in other records.

Adabas Record Design

Once an Adabas file structure has been determined, the next step is usually to define the field definition table for the file. The field definitions are entered as described in the chapter FDT Record Structure. This section describes the performance implications of some of the options which may be used for fields.

The fields should be arranged such that those which are read or updated most often are nearest to the start of the record. This will reduce the CPU time required to locate the fields within the record. Fields which are seldom read but are mainly used as search criteria should be placed last.

The use of groups results in more efficient internal processing of read and update commands. This is the result of shorter format buffers which take less time to process and require less space in the internal format-buffer pool maintained by Adabas.

Numeric fields should be loaded in the format in which they will most frequently be used. This will minimize the amount of format conversion required. However, the relation between CPU time saved and extra disk space required has to be considered.

The use of the fixed storage (FI) option normally reduces the processing time of the field, but may result in a larger disk storage requirement, particularly if the field is contained in a periodic group. NU fields should be grouped together wherever possible. FI fields should be grouped together wherever possible.

It is important that each Adabas base record must fit into one data block, which can be up to 32KB in size (see the section *Container Files* for further information about block sizes). For further information about the space required for the Adabas fields, see the section *Disk Space Usage*. In addition to the space required for the Adabas fields, 4 bytes are needed for each data block and 6 bytes for each Adabas record.

Large Object Values

Normally, each Adabas record must fit into one data block, but there is the exception that values larger than 253 bytes (so-called large object values - LOB values) can be stored by Adabas in a separate internal Adabas file, the associated LOB file. The original Adabas file is called the base file, the records in the base file are called base records, and the records in the LOB file are called LOB records. The following rules apply to LOB values:

- If there is no LOB file associated with an Adabas file, all values are stored in the base record - the maximum field length is then limited to 16381 bytes.
- If a LOB field is a descriptor or a parent field of a derived descriptor, the field values are always stored in the base record, and the maximum field length is limited to 16381 bytes.
- In all other cases, values up to 253 bytes are always stored in the base record, values larger than 253 bytes are stored in one or more LOB segment records in the LOB file - then the base record contains a 16 byte LOB value reference instead.

The following table provides an overview of the terms used in conjunction with LOB values in the Adabas documentation:

Term	Definition
Base file	An Adabas file with a user-defined FDT that contains one or more LOB fields
Base record	A record in a base file. The LOB values in the record are represented by LOB value references pointing to LOB segment records in the LOB file. The actual LOB values are contained in these segment records
LOB	Large Object. Initially, a LOB value in Adabas can have a size of up to ca. 2GB (theoretically)
LOB field	A new type of field in an Adabas file that stores LOB values
LOB file	An Adabas file with a predefined FDT containing LOB values that are spread over one or more LOB segment records
LOB file group	The pair consisting of base file and LOB file, viewed as a single unit
LOB segment	One portion of a LOB value that was partitioned. A LOB value consists of one or more LOB segments
LOB segment record	A record in a LOB file that contains a LOB segment as payload data and other information as control data
LOB value	An instance of a LOB
LOB value reference	A reference or pointer from a base record to the LOB segment records that contain the partitioned LOB value

It is possible to define the following types of LOB fields:

- Binary large objects (BLOBs). These LOB fields are defined with format A (ALPHANUMERIC) and the following options: LB (values > 16381 bytes are supported), NB (blanks at the end of the value are not truncated) and NV (binary values without ASCII / EBCDIC conversion between Open Systems platforms and mainframe platforms).
- Character (ASCII / EBCDIC) large objects (CLOBs). These LOB fields are defined with format A and option LB, but without option NB and NV. In addition, the field options MU, NC, NN, NU can be defined with the usual rules for these options; a LOB field can also be defined as a member of a periodic group.

Notes:

1. Unicode LOB fields (fields with format W (UNICODE)) are currently not supported. You can store large Unicode values in BLOB fields, but then a conversion to other encodings as provided by format W is not supported.
2. For detailed information on the Adabas formats and field options, please refer to the section *FDT Record Structure*.

Spanned Records

From Adabas Version 6.7, records can be spanned in a database. When record spanning is enabled, the size of compressed records in a file may exceed the maximum data storage block size of 32 KB.

Notes:

1. Spanned records must be explicitly allowed for a file. You can do this using the ADADBM function RECORDSPANNING.
2. You can check whether record spanning is enabled for a file with the ADAREP utility.

Adabas System Files

The Adabas nucleus uses some internal Adabas files, the so-called Adabas system files, to store internal data. While the checkpoint file, the security file and the user data file are always required for an Adabas database and are defined when the database is created, the replication system files are only created when you initialize the Adabas-to-Adabas replication - please refer to the section *Adabas-to-Adabas (A2A) Replication* for further details.The RBAC system file is only created when you define it explicitly - please refer to the section *Authorization for Adabas Utilities* for further details.

The following Adabas system files exist:

- Checkpoint File
- Security File
- User Data File
- Replication System Files
- RBAC System File

Checkpoint File

The Adabas checkpoint file is used to log some important events, the Adabas checkpoints; these checkpoints are written for:

- Adabas utility executions
- Nucleus starts and stops
- Adabas user sessions with exclusive access to Adabas files
- User-defined checkpoints

The checkpoints can be displayed with the utility ADAREP parameter CHECKPOINTS. The documentation of this parameter contains a description of the different checkpoint types.

The checkpoints are especially important for the utility ADAREC (database recovery), which re-applies all database updates performed after a database backup: because ADAREC cannot recover some utility operations, it stops when it detects a SYNP checkpoint that indicates the execution of a utility which cannot be recovered by ADAREC, and which must be re-executed manually.

Note: The information stored in the checkpoint file does not contain all of the information required to re-execute the utilities. Software AG therefore strongly recommends that you document all utility executions in order to be able to recover the database if necessary.

Security File

The Adabas security file contains the Adabas security definitions. For more information on Adabas security, For further information, please refer to the *documentation of the ADASCR utility*, which is used to maintain the Adabas security definitions.

User Data File

The user data file is used to store information about the last transaction for all User IDs (ETIDs) specified in the Additions1 field for an OP command. The idea of specifying ETIDs is to enable the implementation of restart processing for programs using Adabas, following a crash. If you don't do this, it doesn't make sense to specify ETIDs.

Notes:

1. Software AG recommends that you only use ETIDs if you really intend to store data in the user data file.
2. If you use Natural to access your Adabas database, please refer to the Natural documentation for more information how to use ETIDs with Natural.
3. Software AG recommends that you do not use the process ID as ETID (for example by specifying “$$” as ETID in Natural); you can only access the user data if you know the process ID of the process which generated the user data, and on some operating systems the process ID can become very long, and as a consequence you can get a very large number records in the user data file.

You can delete the data in the user data file by using the ADADBM REFRESH function; of course then all user data stored in the file are lost.

Replication System Files

Please refer to the section *Adabas-to-Adabas (A2A) Replication* for detailed information about the replication system files.

RBAC System File

Please refer to the section *Adabas Role-Based Security (ADARBA)* for detailed information about the RBAC system file.

Data Access Strategies

Efficient Use of Descriptors

Descriptors are fields for which Adabas has created an index for efficient search operations, to control a logical sequential read and as a sort key in certain Adabas commands such as FIND and SORT ISN LIST. The use of descriptors is, therefore, closely related to the access strategy to be used for a file. Additional disk space and processing overhead is required for each descriptor, particularly those which are updated frequently. The following guidelines may be used in determining the number and type of descriptors to be defined for a file:

- The value distribution of the descriptor should be such that it may be used to select a small percentage of the total number of records in the file;
- Additional descriptors should not be defined to further refine search criteria if a reasonably small number of records can be selected using existing descriptors;
- If two or three descriptors are used in combination frequently (for example, area, department, branch), a superdescriptor may be used instead of defining separate descriptors;
- If the selection criterion for a descriptor always involves a range of values, a subdescriptor may be used;
- If the selection criterion for a descriptor never involves the selection of null values, and a large number of null values are possible for the descriptor, the descriptor should be defined with the null value suppression option;
- If a field is updated very frequently, you should be aware that the faster search operations achieved by using the index created for a descriptor have to be paid for by slower database update operations, since the index also has to be updated;
- If you don't define a field as a descriptor and you perform a non-descriptor search, you should be aware that the performance may be good with a small test database, but that it may be poor with a large production database.
- If you store descriptor values larger than 253 bytes in the database, they are stored in index blocks with a block size of at least 16 KB. If you want to store such values, the database must

have an appropriate ASSO container file or there must be a location defined so that the database can create such an ASSO container.

- The maximum length of a descriptor value is 1144 bytes. Normally, database operations that attempt to insert a larger value will be rejected, but if you specify the index truncation option for the descriptor, larger index values are truncated. The consequence of this is that the results of search operations may no longer be exact if truncated index values are involved; a warning is issued in such cases.

Superdescriptor

A superdescriptor is a descriptor which is created from a combination of up to 20 fields (or portions thereof). The fields from which a superdescriptor is derived may or may not be descriptors. Superdescriptors are more efficient than combinations of ordinary descriptors when the search criteria involves a combination of values. This is because Adabas has only to access one inverted list instead of several and does not have to AND several ISN lists to produce the final list of qualifying records. Superdescriptors may also be used in the same manner as ordinary descriptors to control the logical sequence in which a file is read and to sort ISN lists.

The values for search criteria which use superdescriptors must be provided in the format of the superdescriptor (binary for superdescriptors derived from all numeric fields, otherwise alphanumeric). If the superdescriptor format is binary, the input of the search value using an ad hoc query or report facility may be difficult.

Subdescriptor

A subdescriptor is a descriptor which is derived from a portion of a field. The field used to derive the subdescriptor may or may not be a descriptor. If a search criteria involves a range of values which is contained in the first n bytes of an alphanumeric field or the last n bytes of a numeric field, a subdescriptor derived from the relevant bytes of the field may be defined. This will enable the search criterion to be represented as a single value rather than a range which will, in turn, result in more efficient searching since Adabas will not need to merge intermediate ISN lists. For example, assume an alphanumeric field AREA of 8 bytes, the first 3 of which represent the REGION and the last 5 the DEPARTMENT. If only records for REGION 111 are desired, a search criteria of AREA = 11100000 through 11199999 would be required. If the first three bytes of AREA were defined as a subdescriptor, a search criteria equal to REGION = 111 could be specified.

Phonetic Descriptor

A phonetic descriptor may be defined to perform phonetic searches. The use of a phonetic descriptor in a FIND command results in the return of all the records which contain similar phonetic values. The phonetic value of a descriptor is based on the first 20 bytes of the field value with only alphabetic values being considered (numeric values, special characters and blanks are ignored).

Hyperdescriptor

The hyperdescriptor enables descriptor values to be generated based on a user-supplied algorithm coded in a hyperexit. Up to 255 different user-written hyperexits can be defined for a single Adabas database, and each hyperexit can handle multiple hyperdescriptors.

Hyperdescriptors can be used to implement n-component superdescriptors, derived keys, or other key constructs. See *FDT Record Structure* and *User Exits and Hyperexits* in this manual for more information about hyperdescriptors.

Soft Coupling

A multi-file query may be performed by specifying a field to be used for inter-file linkage in the search criteria. This feature is called soft coupling and does not require the files to be physically coupled.

ISNs

Each record in an Adabas file has an internal sequence number (ISN), which is a 4 byte unsigned integer >0. ISNs are used by Adabas internally to perform queries efficiently, and the result of Adabas FIND commands represents the result as an ISN list.

If the ISN of a record is known, it is very efficient to access the record via its ISN (Adabas L1 command).

User-Assigned ISNs

The user has the option of assigning the ISN of each record in a file rather than having this done by Adabas. This technique permits subsequent retrieval using the ISN directly rather than using the inverted lists. This requires that the user develop his own algorithm for the assignment of ISNs. The resulting range of ISNs must be within the space allocated for the Address Converter for the file (please refer to the description of the MAXISN parameter in the chapter ADAFDU in the Utilities Manual for more information), and each application which adds records to the file must contain the user's ISN assignment algorithm.

Using ISNs as a Descriptor

The user may store the ISNs of related records in another record in order to be able to read the related records directly without using the inverted lists.

For example, assume an application needs to read an order record and then find and read all customer records for the order. If the ISN of the customer record (if there is more than one customer per order, a multiple-value field could be used) were stored in the order record, the customer record could be read directly since the ISN is available in the order record. This would avoid the necessity of issuing a FIND command to the customer file to determine the customer records for the order. This technique requires that the field containing the ISNs of the customer records be established and maintained in the order record, and assumes that the ISN assignment in the customer file will not be changed as a result of a file unload and reload in which the same ISN assignment is not retained.

ADAM Usage

The Adabas Direct Access Method (ADAM) facility permits the retrieval of records directly from Data Storage without access to the inverted lists. The Data Storage block number in which a record is located is calculated using a randomizing algorithm based on the ADAM key of the record. The use of ADAM is completely transparent to application programs.

The main performance advantage of using ADAM descriptors is the reduction in the number of accesses made to inverted lists. The main advantage of using an ISN as an ADAM key is the reduction in the number of accesses to the Address Converter.

ADAM will generally use an average of 1.2 to 1.5 (logical) I/O operations (including an average of overflow records stored under Associator control in other blocks of the file) to search for a record via the Adam key, as opposed to the three to four I/O operations required to search fora record using the inverted lists. Overflow records are also retrieved using normal Associator inverted-list references.

The ADAM key of each record must be a unique value. The ISN of a record may also be used as the ADAM key.

Notes:

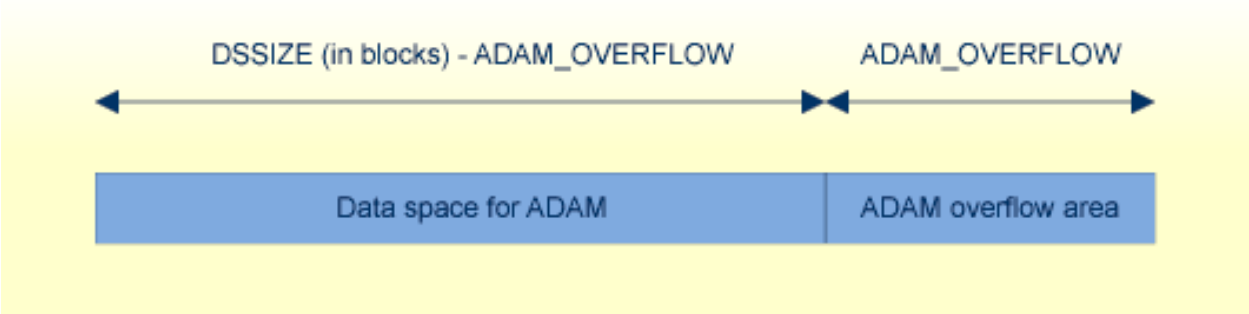
1. Access to ADAM files via the ADAM key is only efficient if the file was defined with sufficent size to avoid having too many records being stored in the overflow area. If a record did not fit into the block determined by the hash algorithm and searching the record in this block was not successful, the normal search algorithm is performed afterwards. In this case, access to the record requires more time than is required for non-ADAM files.
2. In order to avoid block overflows and the resulting records in the overflow area, you should normally define ADAM files with enough size such that the average fill factor of a block is relativly small. This means that utilities will generally require more time than for non-ADAM files with the same records, because they must perform more I/Os. This applies in particular to

ADAMUP if the sort sequence of the records to be loaded is not the sequence of the hash keys, because then the next record to be loaded, in most cases, belongs to a different block.

The file definition utility ADAFDU is used to define a descriptor or ISN as an ADAM key. There are 3 parameters:

Parameter	Meaning
ADAM_KEY	define ADAM key
ADAM_PARAMETER	parameter to influence data record distribution algorithm
ADAM_OVERFLOW	number of overflow blocks

The data space for ADAM is calculated as (DSSIZE (in blocks) - ADAM_OVERFLOW):



The data space for ADAM cannot be subsequently extended, only the ADAM overflow area can grow. However, the ADAM area can cover multiple DS extents within the initialization. The ADAM area is formatted and marked as in use during the execution of ADAFDU.

The number of blocks to be used for the overflow area is defined with the ADAM_OVERFLOW parameter. A minimum of one block is required, and more blocks can be added later. The overflow blocks are used if there is no space for the ADAM-calculated block for a new record. The gain in performance obtained by using ADAM is decreased if a large number of records is stored in the overflow area. The distribution of records in the ADAM file can be checked using the file information utility ADAFIN.

If the space reusage option has been set for the file, it only applies to the overflow area. The DATA padding factor applies to both areas (DATA and overflow).

The ADAM_PARAMETER parameter is used to influence the distribution of the data records.

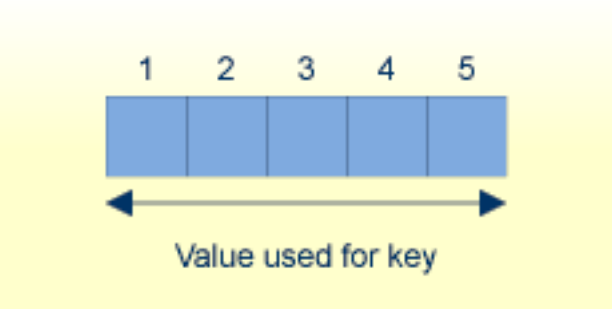
If the ADAM key is ISN or a fixed-point descriptor, it determines the number of consecutive values that are to be stored in one block. The basic algorithm is

```
DS number = (actual value/ADAM_PARAMETER) modulo number_adam_blocks
```

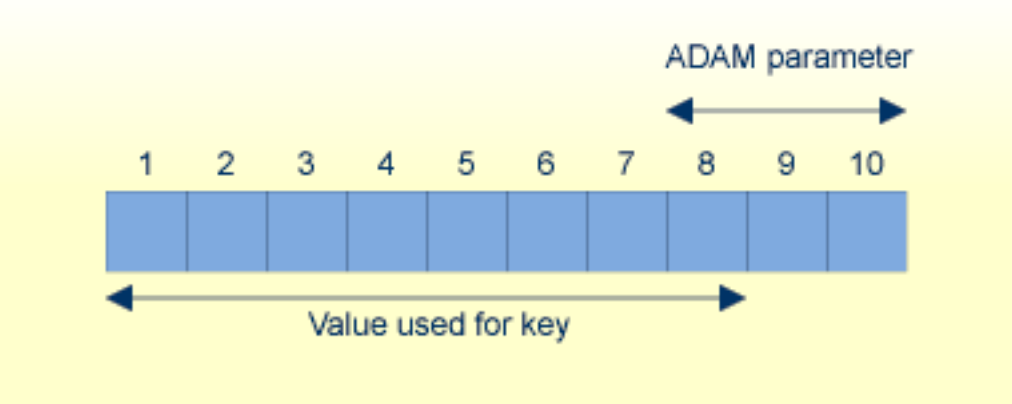
If the format of the ADAM key is alphanumeric, binary or floating point, then the ADAM parameter defines the offset from the end of the value for an 8-byte extraction.

Example: ADAM key with format A:

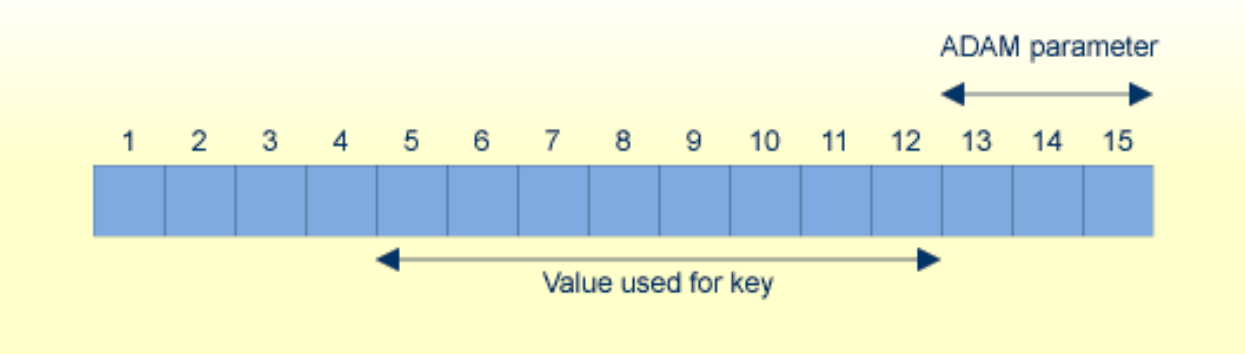
```
ADAM_PARAMETER = 3
Value's lengths = 5, 10, 15
```



If the value is less than or equal to 8 bytes long, the complete value is taken as the extraction value.



Otherwise, if (value length - ADAM parameter) is less than or equal to 8 bytes long, the first 8 bytes are taken as the extraction value.



Otherwise the last 8 bytes after removing (ADAM parameter) bytes are taken as the extraction value.

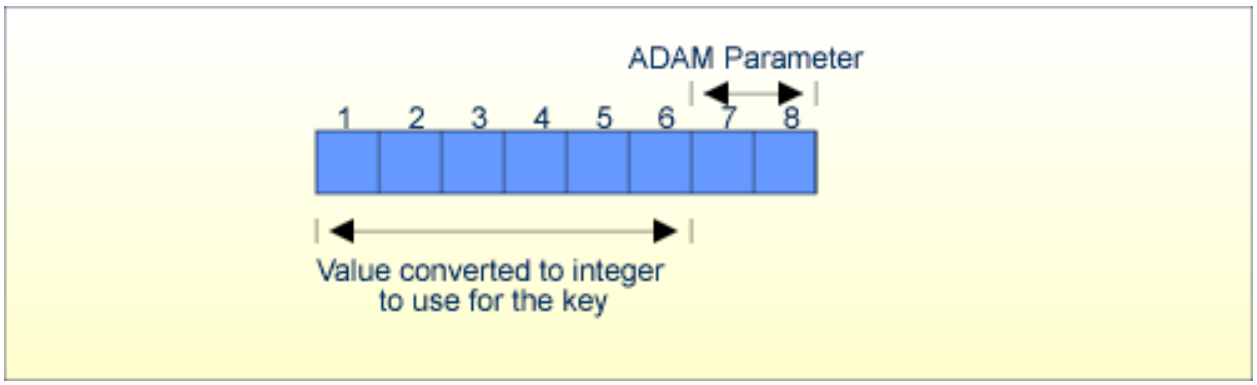
The algorithm for calculating a relative DS number is:

```
DS number = (extraction value) modulo (number of ADAM blocks)
```

If the format of the ADAM key is packed or unpacked, then the ADAM parameter defines the offset from the end of the value to the position of the value to be considered for the ADAM value. This ADAM value from position 1 to the position (value length - ADAM_parameter) will be converted to a 8-byte integer value.

Example: ADAM key with format P or U

```
ADAM_PARAMETER = 2
Value's length = 8
```



The algorithm for calculating a relative DS number is:

```
DS number = (integer value) modulo (number of ADAM blocks)
```

If the format of the ADAM key is fixed point or if the ADAM key is the ISN, the extraction value is (ADAM key value) / (ADAM parameter).

Example: ADAM key with format F

The following values are entered in ADAFDU:

```
DSSIZE = 40 B
ADAM_KEY = FF
ADAM_OVERFLOW = 10
ADAM_PARAMETER = 12
```

The file is an ADAM file, and the unique descriptor FF is used as an ADAM key. 30 blocks will be used for the ADAM DS area, with 10 blocks reserved for the overflow area. 12 consecutive values will be stored in each block.

The values will be stored in the DS blocks as follows:

FF Value	DS Block
0 - 11	1
12 - 23	2
24 - 35	3
...	...
348 - 359	30
360 - 371	1
372 - 383	2
...	...

Disk Space Usage

The efficient use of disk space is especially important in a database environment because:

- The sharing of data between several users, possibly concurrently and in different combinations, normally requires that a large proportion of an organization's data be stored online;
- Some applications contain extremely large amounts of data.

Decisions concerning the efficient usage of disk space must be made while considering other objectives of the system (performance, flexibility, ease of use). This section discusses the techniques and considerations involved in making trade-offs between these objectives and the efficient usage of disk space.

Compression

Each field may be defined to Adabas with one of three compression options:

1. Ordinary compression (the default) which causes Adabas to remove trailing blanks from alphanumeric fields and leading zeros from numeric fields, but requires one additional length byte if the compressed value length is <= 126, or two if the compressed value length is larger. The null value is compressed to a length byte = 1.
2. Null value suppression which results in ordinary compression and, in addition, suppresses the null value for the field.
3. Fixed storage (FI), in which the field is not compressed at all, but the additional length byte in Data Storage is omitted.

The figure Adabas Compression below illustrates how various values of a five-byte alphanumeric field are stored using each compression option. The number preceding each stored value is an inclusive length byte (not used for FI fields). The asterisk shown under null value suppression indicates a suppressed field count. This is a one-byte field which indicates the number of empty (suppressed) fields present at this point in the record. A `b' means a blank.

```
Field     Ordinary      Fixed      Null Value
Value     Compression   Storage    Suppression

ABCbb     04414243      4142432020 04414243
          (4 bytes)     (5 bytes)  (4 bytes)

ABCDb     0541424344    4142434420 0541424344
          (5 bytes)     (5 bytes)  (5 bytes)

ABCDE     064142434445  4142434445 064142434445
          (6 bytes)     (5 bytes)  (6 bytes)
bbbbb     01            2020202020  *
          (2 bytes)     (5 bytes)  (1 byte)
```

```
Field     Ordinary      Fixed      Null Value
Value     Compression   Storage    Suppression

ABCbb     4ABC          ABCbb      4ABC
          (4 bytes)     (5 bytes)  (4 bytes)

ABCDb     5ABCD         ABCDb      5ABCD
          (5 bytes)     (5 bytes)  (5 bytes)

ABCDE     6ABCDE        ABCDE      6ABCDE
          (6 bytes)     (5 bytes)  (6 bytes)
bbbbb     1b            bbbbb      *
          (1 byte)      (5 bytes)  (1 byte)
```

The compression options chosen also affect the creation of the inverted list for the field (if it is a descriptor) and the processing time needed for compression and decompression of the field.

Fixed Storage

Fixed storage indicates that no compression is to be performed on the field. The field is stored according to its standard length with no length byte. Fixed storage is useful for small fields and for fields for which little or no compression is possible. See *FDT Record Structure* for information about the various restrictions related to the use of FI fields.

Ordinary Compression

Ordinary compression results in the removal of trailing blanks from alphanumeric fields and leading zeros from numeric fields. As can be seen in the figure Adabas Compression above, ordinary compression will result in a saving in disk space if at least 2 bytes of trailing blanks or leading zeros are removed.

Null Value Suppression

If null value suppression is specified for a field, and the field value is null, a one-byte empty field indicator will be stored instead of a length byte and the compressed null value (see figure above). This empty field indicator specifies the number of consecutive null-value suppressed fields which contain null values at this point in the record. Up to 63 empty fields can be represented by one byte. It is, therefore, advantageous to physically position fields which are frequently empty next to one another in the record and to define each with the null-value suppression option.

If the field is a descriptor, the use of null value suppression will result in the omission of the null value from the inverted lists. This means that a FIND command, in which the null value of the descriptor is used will always result in no qualifying records even if there are records in Data Storage which contain a null value for the descriptor. This applies also to subdescriptors and superdescriptors derived from a field defined with null value suppression. No entry will be made for a subdescriptor if the bytes of the field from which it is derived contain a null value and the field is defined with the null-value suppression option. No entry will be made for a superdescriptor if the bytes of any of the fields from which it is derived contain a null value and the field is defined with the null-value suppression option.

The use of null value suppression with descriptor fields, therefore, depends on the need to search for null values, and, if the descriptor is used to control logical sequential reading or sorting, the need to read records containing a null value. If such a need does not exist, null value suppression is normally used (unless the FI option is used).

Null value suppression is normally recommended for multiple-value fields and fields within periodic groups in order to reduce both the amount of disk space required and the internal processing requirements of these types of fields. The updating of such fields varies according to the compression option used. If a multiple-value field defined with null value suppression is updated with a null value, all values to the right are shifted left, and the value count is reduced accordingly. If all the fields of a periodic group are defined with null value suppression, and the entire group is updated to a null value, the occurrence count will be reduced only if the occurrence updated is the highest (last) occurrence. For detailed information about the updating of multiple-value fields and periodic groups, see *FDT Record Structure* and the *Command Reference*, *A1 command* and *Command Reference*, *N1/N2 command*.

Multiple-Value Fields and Periodic Groups

The values for multiple-value fields and periodic groups are normally preceded by an 8-byte header (or sometimes by a one byte MU or PE count). Each occurrence of a periodic group is preceded by a two-byte length indicator. If a periodic group contains empty occurrences, up to 32767 empty occurrences are compressed to a 2-byte empty periodic group occurrence counter.

Padding Factors

A large amount of record update activity (A1 command) may result in a considerable amount of record migration, i.e. moving the record from its current block to another block in which there is sufficient space for the expanded record. Record migration may be considerably reduced by defining a larger padding factor for Data Storage for the file when it is loaded. The padding factor represents the percentage of each physical block that is to be reserved for record expansion. The padding area is not used during file loading or when adding new records to a file. A large padding factor should not be used if only a small percentage of the records are likely to expand, since the padding area of all the blocks in which non-expanding records are located would be wasted.

If a large amount of record update/addition is to be performed, in which a large number of new values must be inserted into the current value range of one or more descriptors, a considerable amount of migration may also occur within the Associator. This may be reduced by assigning a larger padding factor for the Associator.

The disadvantages of a large padding factor are a larger disk-space requirement (less records or entries per block) and possible degradation of sequential processing, since more physical blocks will have to be read.

Padding factors are specified when a file is defined (using utility ADAFDU) and can be changed (using utility ADAORD).

Security Planning

This section describes the general considerations which should be made concerning database security and explains the Adabas facilities which may be used to secure data contained within the database.

Effective security measures must take account of the following points:

- A system is only as secure as its weakest component. This may be a non-DP area of the system: for example, failure to properly secure printed listings;
- It is rarely possible to design a 'foolproof' system. A security violation will probably be committed if the gain is likely to exceed the cost;
- Security costs can be high. These costs include inconvenience, machine resources and the time required to coordinate the planning of security measures and monitor their effectiveness.

The cost of security measures is usually much easier to quantify than the risk or cost of a security violation. The calculation may, however, be complicated by the fact that some security measures may offer benefits outside the area of security. The cost of a security violation depends on the nature of the violation. Possible types of cost include:

- Loss of time while the violation is being recovered;

- Penalties under privacy legislation, contracts, etc.;
- Damage in relationships with customers, suppliers, etc.

Adabas Security Facilities Overview

This section contains an overview of the security facilities provided by Adabas and its subsystems. For more detailed information about the facilities discussed in this section, please refer to the section *Adabas Security Facilities*.

Adabas Authentication

Adabas Authentication provides a means for applications to access the database in the context of a user by having the user provide valid credentials.

For more detailed information about Adabas Authentication, please refer to *Adabas Authentication* in the section *Adabas Security Facilities*.

Authorization for Adabas Utilities

Authorization for Utilities provides a means of restricting the usage of Adabas utilities on databases by assigning users a role which has selective access privileges.

For more detailed information about Authorization for Adabas Utilities, please refer to *Authorization for Adabas Utilities* in the section *Adabas Security Facilities*.

Adabas Password Security (ADASCR)

For more detailed information about the Adabas Password Security (ADASCR), please refer to *Adabas Password Security* in the section *Adabas Security Facilities*.

Ciphering

Ciphering prevents the unauthorized analysis of Adabas container files. Adabas can cipher the data that it stores in container files. This, however, only applies to the data records that are stored in the Data storage, but not to the inverted lists on the Associator.

For more detailed information about ciphering, please refer to *Ciphering* in the section *Adabas Security Facilities*.

Transaction Concept

An important concept for all databases is the availability of a transaction concept in order to guarantee database integrity. A transaction guarantees that a set of database update operations will either be committed, i.e. they all the updates become persistent in the database, or in case of an error, the update operations already performed will be completely be rolled back.

This section is just a short overview on the transaction concept; please refer to the *Command Reference* section for further information.

Adabas has the following database commands to support the transaction concept:

- ET - End of Transaction, commits a transaction
- BT - Backout Transaction, rolls back a transaction.

For some files, it can be desirable that they do not take part in normal transaction logic, and that all database modifications for the file are kept in the database even if a transaction is rolled back. An example for such a file is a log file, in which all activities of a user are to be logged including activities within a transaction that is later backed out.

Lock Concept

In order to guarantee database integrity, it must not be possible for another user to update records that required for a transaction. To this end, Adabas lets you lock records for the duration of a transaction.

Adabas supports the following types of lock:

Lock Type	Usage
Share or read lock (S)	You can acquire a shared or read lock if no other user has already acquired an exclusive lock for the record. S locks allow you to guarantee that nobody else can update one or more of the records as long as you have these records locked, while other users can still also get a shared lock.
Exclusive or write lock (X)	You can only acquire an exclusive or write lock for an Adabas record if no other user has already acquired an S or X lock for this record. Modification or deletion of a record is only possible with an X lock of the record. If you create a new record, this record is automatically locked exclusively.

Subtransactions

Sometimes it is can be necessary to roll back not the complete transaction, but only a subset of the transaction. To this end, Adabas has a subtransaction concept, which is implemented via special options of the ET and BT commands.

ET Synchronization

Sometimes it is necessary for the database to be in a consistent state:

- If you create a backup of the database with ADABCK;
- If you perform an external backup with ADAOPR EXT_BACKUP;
- If you switch to a new protection log (PLOG).

Note: Switching to a new PLOG extent does not require ET synchronization, because all extents of a PLOG are considered as one PLOG.

In all of these cases, an ET synchronization must be performed for the database - this means:

- All currently-active transactions continue to work until the transaction is terminated or until a timeout occurs. The timeout period is defined by the Adabas nucleus parameter TT or, in case of ADABCK, by the parameter ET_SYNC_WAIT;
- If an Adabas command tries to start a new transaction, the command has to wait until the ET synchronization is finished;
- As soon as all active transactions are terminated, phase 2 of the ET synchronization begins: all activities to be done in the consistent state can be done, for example a buffer flush, in order to ensure that the consistent state of the database is stored on disk;
- Once phase 2 of the ET synchronization has finished, any commands waiting for the end of the ET synchronization can continue.

If you create a backup on the file level using ADABCK DUMP without the option NEW_PLOG, an ET synchronization is only performed on the file level:

- The ET synchronization waits only for the termination of transactions that access the dumped files;
- New transactions can start as long as they only access other files. However, as soon as such a transaction is extended to one of the dumped files, the command has to wait until the end of the ET synchronization.

Recovery/Restart Design

This section discusses the design aspects of database recovery/restart.

Correct recovery/restart planning is an important part of the design of the system, particularly one in which a database is used. Most of the causes of failure can be anticipated, evaluated and resolved as part of the basic system design process.

Recoverability is often an implied objective. Everyone assumes that, regardless of what happens, the system can be recovered and restarted. There are, however, specific facts to be determined about the level of recovery needed by the various users of the system. Recoverability is an area where the DBA has to take the initiative and establish necessary facts. Initially, each potential user of the system should be asked about their recovery/restart requirements. The most important considerations are:

- How long can the user manage without the system?
- How long can each phase be delayed?
- What manual procedures, if any, does the user have for checking input/output and how long do these take?
- What special procedures, if any, need to be performed to ensure that data integrity has been maintained in a recovery/restart situation?

Planning and Incorporating Recoverability

Once the recovery/restart requirements have been established, the DBA can proceed to plan the measures necessary to meet these requirements. The methodology provided in this section may be used as a basic guideline.

1. A determination should be made as to the level and degree to which data is shared by the various users of the system.
2. The recovery parameters for the system should be established. This includes a predicted/actual breakdown rate, an average delay and items affected, and items subject to security.
3. An outline containing recovery design points should be prepared. Information in this outline should include:
   - Validation planning. Validation of data should be performed as close as possible to its point of input to the system. Intermediate updates to data sharing the record with the input will make recovery more difficult and costly;
   - Dumps (back-up copies) of the database or selected files;
   - User and Adabas checkpoints;
   - Use of ET logic, exclusive file control, ET data.

The recovery strategy should be subjected to all possible breakdown situations to determine the suitability, effectiveness, and cost of the strategy.

4. Operations personnel should be consulted to determine whether all resources required for recovery/restart can be made available if and when they are needed.
5. The final recovery design should be documented and reviewed with users, operations personnel and any others involved with the system.

Locations of Database Containers, Backup Files, and Protection Logs

When you restart the database after a database crash, an autorestart is performed: All transactions that were active when the nucleus crashed are rolled back, and all missing database updates are written to the ASSO and DATA containers. For this purpose, the update operations have been logged on the WORK container. Nevertheless, in case of a disk corruption, it may be that the autorestart fails. In this case, it is important that you can recover the state of your database from a backup and the protection logs. This can be guaranteed only if your backup files and protection logs (PLOGs) are stored on separate, independent disks. Note that Adabas logs the update operations twice: once on WORK for the autorestart, and once on PLOGs for restore/recover to enable the recovery of the current database state in case a disk where a log is stored becomes corrupted.

Notes:

1. In order to avoid disk problems, you can also consider hardware-based solutions such as RAID systems. Nevertheless, it is recommended that you create backups and PLOGs, because a RAID system doesn't protect you from software or handling errors.
2. You should also consider disaster recovery: what do you do if your complete computer system should be destroyed?
3. Today's computers usually have sufficient memory to allow you to use a sufficiently large buffer pool, so that the access characteristics of the ASSO and DATA containers are, in most cases, no longer performance-relevant. An exception to this are large databases, for which it is not possible to define a sufficiently large buffer pool in order to reduce the number of physical I/Os sufficiently.
4. Especially in cases where there are many short transactions with only one or a few update operations, you should use disks with short latency times for the WORK container and the PLOGs, for example SSD disks or storage devices with a cache. The reason for this is that a transaction can only be committed after the log information has been written to WORK and PLOG. Depending on the devices used for PLOG and WORK, a factor of up to 10 or more in the throughput of an application performing a large number of short update transactions has been observed.

Matching Requirements and Facilities

Once the general recovery requirements have been designed, the next step is to select the relevant Adabas and non-Adabas facilities to be used to implement recovery/restart. The following sections describe the Adabas facilities related to recovery/restart.

Transaction Recovery

Many batch-update programs process streams of input transactions that have the following characteristics:

- The transaction requires the program to retrieve and add, update, and/or delete only a few records. For example, an order entry program may retrieve the customer and product records for each order, add the order and order item data to the database, and perhaps update the quantity on order field of the product record;
- The program needs exclusive control of the records it uses from the start of the transaction to the end, but can release them for other users to update or delete once the transaction is complete;
- A transaction must never be left incomplete, i.e. if it requires two records to be updated, either both or neither must be changed.

The Adabas End Transaction (ET) Command

The use of the Adabas ET command will:

- Ensure that all the adds, updates, and/or deletes performed by a completed transaction are applied to the database;
- Ensure that all the effects of a transaction which is interrupted by a total or partial system failure are removed from the database;
- Allow the program to store user-restart data (ET data) in an Adabas system file. This data may be retrieved on restart with the Adabas OP or RE commands;
- Release all records placed in hold status while processing the transaction.

Adabas Close (CL) Command

The Adabas CL command can be used to update the user's current ET data (for example, to set a job completed flag).

Reading ET Data

A user may retrieve his ET data after a user restart or at the start of a new user or Adabas session with the Adabas OP command. The user is required to provide a user identification (USERID) with the OP command. This USERID is used by Adabas to locate the user's ET data.

Another user's ET data may be read by using the RE command, provided that the USERID of the other user is known. This may be useful, for example, for staff supervision of an online update operation.

System or Transaction Failure

In the event of an abnormal termination of an Adabas session, the Adabas AUTOBACKOUT routine, which is automatically invoked at the beginning of every Adabas session, will remove the effects of all interrupted transactions from the database.

If an individual transaction is interrupted, Adabas will automatically remove all the changes the transaction has made to the database. An application program can explicitly cause its current transaction to be backed out by issuing the Adabas BT command.

No-BT Files

In the case of a nucleus crashing, the following points should be taken into consideration:

- All database modifications for a no-BT file issued before the last ET are applied in the database.
- It is not defined whether database modifications for a no-BT file issued after the last ET are applied in the database or not.

Limitations of Adabas Transaction Recovery

The following limitations of Adabas transaction recovery should be considered:

- The transaction recovery facility cannot function if critical portions of the ASSOCIATOR and/or the data protection area of the Adabas WORK cannot be physically read or have been overwritten. Although this situation should rarely occur, specific procedures should be prepared for such a condition. The section on recovery/restart procedures should be consulted for detailed guidelines on how to recover a physically damaged database;
- The transaction recovery facility recovers only the contents of the database. It does not reposition non-Adabas files, or save the status of the user program;
- It is not possible to backout the effects of a specific user's transactions, because other users may have performed subsequent transactions using the records added or updated by the first user.

Adabas Checkpoint Command

Some programs cannot conveniently use ET commands because:

- The program would have to hold large numbers of records for the duration of each transaction. This would increase the possibility of a deadlock situation (Adabas automatically resolves such situations by backing out the transaction of one of the two users, but a significant amount of transaction reprocessing could still result), and a very large Adabas hold queue would have to be established and maintained;
- The program may process long lists of records found by complex searches and restarting part of the way through such a list may be difficult.

Such programs can use the Adabas checkpoint command (C1) to establish a point at which the file or files the program is updating can be restored if necessary. The specific command used depends on the type of updating (exclusive control) being performed.

Exclusive File Control

A user can request exclusive update control of one or more Adabas files. Exclusive control is requested with the OP command and will be given only if the file is not currently being updated by another user. Once exclusive control is assigned to a user, other users may read but not update the file.

Programs which read and/or update long sequences of records, either in logical sequence or as a result of searches, may use exclusive control to prevent other users from updating the records used. This avoids the necessity of issuing a record hold command for each record.

Checkpointing Exclusive Control Files

Exclusive control users may or may not use ET commands. If ET commands are not used, checkpoints can be taken by issuing a C1 command (if user data is to be stored).

System or Program Failure

In the event of a system or program failure, the file or files being updated under exclusive control may have to be restored (using ADABCK or ADAMUP) to the state before the start of the execution of the program which failed.

Limitations of Exclusive File Control

The following limitations apply to exclusive file control:

- Recovery to the last checkpoint is not automatic, and the data protection log in use when the failure occurred is required for the recovery process. This does not apply if the user issues ET commands;
- In a restart situation following a system failure, Adabas does not check nor prevent other users from updating files which were being updated under exclusive control at the time of the system interruption.

User Restart Data

The Adabas ET and CL commands provide an option of storing up to 2000 bytes of user data in an Adabas system file.

The Adabas ET, C3 and CL commands provide an option of storing up to 2000 bytes of user data in an Adabas system file.

One record of user data is maintained for each user. This record is overwritten each time new user data is provided by the user. The data is maintained across Adabas sessions only if the user provides a user identification (USERID) with the OP command. User data may be read with an OP or RE command. A user may read another user's data with the RE command, provided that the USERID of the other user is known.

The primary purpose of user data is to enable programs to be self-restarting and to check that recovery procedures have been properly carried out. The type of information which may be useful as user data includes:

- The date and time of the original program run and the time of last update. This will permit the program to send a suitable message to a terminal user, console operator or printer to allow the user and/or operator to check that recovery and restart procedures have operated correctly. In particular, it will allow terminal users to see if any work has to be rerun after a serious overnight failure that they were not aware of;
- The date of collection of the input data;
- Batch numbers. This will enable supervisory staff to identify and allocate any work that has to be reentered from terminals;
- Identifying data. Sometimes this may simply be a supplement to or cross-check on the items described above. In other cases, it may be the principal means of deciding where to restart, e.g., a program driven by a logical sequential scan needs to know the key value at which to resume;
- Transaction number/input record position. This may allow a terminal user or batch program to locate the starting point with the minimum of effort. Although Adabas returns a transaction sequence number for each transaction, the user also may want to maintain a sequence number because:

- After a restart, the Adabas sequence number will be reset;
- If transactions vary greatly in complexity, there may not be a simple relationship between the Adabas transaction sequence number and the position of the next input record or document;
- If a transaction is backed out by the program because of an input error, Adabas does not know whether the transaction will be reentered immediately (it may have been a simple keying error) or rejected for later correction (if there was a basic error in the input document or record);

- Other descriptive or intermediate data. For example, totals to be carried forward, page numbers and headings of reports, run statistics;
- Job/batch completed flag. The system may fail after all processing has been completed but before the operator or user has been notified. In this case, the operator should restart the program which will be able to check this flag without having to run through to the end of the input. The same considerations apply to batches of documents entered from terminals;
- Last job/program name. If several programs must update the database in a fixed sequence, they may share the same USERID and use user data to check that the sequence is maintained.

3 Container Files

General

Container files are disk files created by Adabas utilities. They are managed by the Adabas nucleus and Adabas utilities. The internal structure of these files is organized and maintained by Adabas, thus permitting the use of very efficient disk usage algorithms.

The data in the container files consists of data blocks with a block size that is defined by the creator of the database. All of the data blocks of each container type are addressed via a so-called relative adabas block number (RABN), which is a 4-byte unsigned integer >0. Therefore an Adabas database can contain up to 2^{32} - 1 blocks of each container type. The term RABN is used not only for the block number, but also for the corresponding block.

The required container files of an Adabas database are called ASSO, DATA and WORK. For some utilities, additional container files called SORT and TEMP are required.

Associator (ASSO)

ASSO contains the organizational data of the database and of the files in the database. Examples of the data stored in ASSO are:

- a summary of the physical and logical layout of the database.
- a list of the used and unused blocks of the database.
- a description of the record fields of each file.
- lists of descriptor (search key) fields, which are used for non-sequential database search operations.
- protection mechanisms for using the Adabas utilities when the database is offline.

Data Storage (DATA)

Data Storage (also referred to as simply DATA) contains the user data of a database. In order to reduce disk space requirements, Adabas uses a data compression technique. This means that user data is converted into a more compact form before being stored in DATA, thus significantly reducing storage requirements and disk I/O.

WORK

The Adabas nucleus uses WORK as a temporary storage area for update log information required for backout transaction and auto restart.

The size of the WORK should be chosen such that the following applies at all times: consider all of the update, delete and store operations performed since the start of the oldest transaction that is currently active - then the size of the WORK should be equal to or greater than

- (the size of all old compressed records modified or deleted

- + the size of all new compressed records after modification or insertion
- + the size of all old index values modified or deleted
- + the size of all new index values after modification or insertion)

multiplied by 4.

Note: Databases with LOB data may imply significantly larger WORK sizes because the size of the LOB data also has to be taken into account (for updated records, only the size of the LOB values which are updated). If a database contains LOB data, a WORK block size of 4KB is recommended.

SORT, TEMP

These are used by some Adabas utilities as temporary storage areas and work areas. In addition to the predefined SORT and TEMP containers, Adabas also uses temporary files created by the nucleus or utilities as work space, these files being deleted after usage. Refer to the section *Temporary Working Space* for further information.

Adabas Logical Extents

An Adabas logical extent is a group of consecutive RABNs allocated by the Adabas nucleus or an Adabas utility.

For each file loaded into the database, at least one of each of the following types of Adabas logical extents is allocated to the file:

- **Data Storage logical extent**
  (allocated from the Data Storage physical extent);
- **Address Converter logical extent**
  (allocated from the Associator physical extent);
- **Normal Index logical extent**
  (allocated from the Associator physical extent);
- **Upper Index logical extent**
  (allocated from the Associator physical extent).

Additional logical extents are allocated by the Adabas nucleus or an Adabas utility when additional space is needed as a result of file updating.

Adabas Physical Extents

Up to 16 extents are allowed for the Associator and the Data Storage: ASSO1 ... ASSO16, DATA1 ... DATA16. Sort and Work may have two extents: SORT1 and SORT2 and WORK1 and WORK2. Temp can have only one extent: TEMP1. These extents have to be defined in consecutive numerical order, i.e. if ASSO1, ASSO2 and ASSO4 are defined, ASSO4 will not be found because ASSO3 is not defined.

Adabas Physical Extents

The datasets ASSO, DATA, WORK, SORT and TEMP can consist of several extents, i.e. physically separate areas of storage on disk or other secondary storage medium. When a utility references any of these extents, it uses environment variables to do so. The environment variables are called ASSO1, ASSO2 etc. for the ASSO dataset, DATA1, DATA2 etc. for the DATA dataset and so on for WORK, SORT and TEMP. Thus, for example, if a utility requires to access the ASSO dataset which has three extents, the environment variables ASSO1, ASSO2 and ASSO3 must point to these extents.

The search strategy for finding the ASSO, DATA and WORK container extents is as follows:

1. Check for the environment variables ASSO1, ASSO2 etc. for ASSO, DATA1, DATA2 etc. for DATA and WORK1 for WORK. If such an environment variable exists, it must contain the file name of the corresponding container extent.
2. Search for the corresponding entries in the DBnnn.INI file. If such an entry exists, it must contain the file name of the container extent. Refer to the *Adabas Extended Operation* documentation for further information about the DBnnn.INI files.
3. Search for the file CONTx.nnn in the database directory (UNIX: $ADADATADIR/dbnnn, Windows: %ADADATADIR%\dbnnn), where CONT is ASSO, DATA or WORK, x is the extent number and nnn is the 3 digit database ID.

The search stategy for using SORT and TEMP is described in the section *Temporary Working Space*

The maximum number of ASSO extents is given by (ASSO1 blocksize - 2) / 12. The maximum the number of DATA extents is given by (ASSO1 blocksize*3 - 2) / 12. These values can, however, be reduced under the circumstances described below.

The total number of ASSO and DATA extents cannot exceed 2721. This maximum number reduces by 1 each time any two adjacent DATA extents have a different block size. So the formula is:

ASSO Extents + DATA Extents + (number of different adjacent DATA block sizes) <= 2721.

Thus, for example, you could have a database where there is only 1 ASSO extent and 1360 DATA extents where no two adjacent DATA extents have the same block size, giving a total of 1 ASSO extent + 1360 DATA extents + 1359 changes of DATA block size= 2720.

The following table gives some examples of the correspondence between the size of the container file ASSO1 and the number of ASSO and DATA extents allowed. The entries in the column "best case" show the maximum number of DATA extents allowed if all of the DATA extents have the same block size. The entries in the column "worst case" show the maximum number of DATA extents allowed if no two adjacent DATA extents have the same block size.

ASSO1 blocksize	max. number of ASSO extents	max number of DATA extents best case	worst case
2 KB (2048 bytes)	170	511	511
3 KB (3072 bytes)	255	767	767
4 KB (4096 bytes)	341	1023	1023
5 KB (5120 bytes)	426	1279	1148
6 KB (6144 bytes)	511	1535	1105
7 KB (7168 bytes)	597	1971	1062
8 KB (8192 bytes)	682	2047	1020

SORT can have up to 50 extents: SORT1, SORT2, ... ,SORT50

WORK can have only 1 extent: WORK1.

TEMP can have up to 10 extents: TEMP1, TEMP2, ... ,TEMP10.

Effect of large buffer sizes for PLOG and WORK

If you specify a WORK block size of 8K or less, Adabas will set the PLOG block size to 8K. If you specify a WORK block size larger than 8K, Adabas will set the PLOG block size to 32K.

To ensure that all completed transactions can be re-applied during a database recovery, the PLOG buffer is flushed to the PLOG after each ET command, regardless of whether the PLOG buffer is full or not. Each subsequent ET command causes the current PLOG block to be re-written, as long as the PLOG buffer is not full. A new PLOG block is only started when the PLOG buffer is full. Similarly, to ensure data consistency after an autorestart, the current WORK part 1 block is re-written after each ET command until the WORK part 1 buffer is full.

In general, if you have large PLOG and WORK block sizes, more transactions are required to fill the PLOG buffer and WORK part 1 buffer than with small block sizes. This means that the average size of the I/O transfers is increased, but the total number of I/O transfers due to ET commands is unchanged.

For this reason we recommend you to use a WORK block size of 8K or less if your compressed data records do not exceed 8K, and therefore a PLOG block size of 8K.

Access Methods for Container Files

Adabas offers two methods for creating and accessing database container files:

- Device type independent access method
- Device type dependent access method

Device type independent access method

With the device type independent access method, Adabas requests the operating system to create the container file using the "contiguous best try" option. The Adabas blocks are written contiguously, regardless of the physical device characteristics. You select the device type independent access method for a given container file by specifying the size of the container file in megabytes when you create it.

With modern hardware (RAID systems, variable track size disks, storage servers, etc.) the track size returned by the system is an arbitrary number and bears no relation to the physical characteristics of the disk. In this case you should use the device type independent access method.

The device type independent access method is always used to access the SORT and TEMP container files.

Device type dependent access method

You select the device type dependent access method for a given container file by creating the container file as a number of contiguous cylinders that start on a cylinder boundary. The values sectors/track and tracks/cylinder that the system returns as device information are used and a cluster size that allows the allocation of one single cylinder is required. In other words, the number of sectors per cylinder must be a multiple of the cluster size.

When a block is written to a container file with this access method, Adabas ensures that the block does not span track boundaries. If the track size is not a multiple of the Adabas block size, the end of the track will not be used. This allows Adabas blocks to be read with a single disk revolution.

Use the OpenVMS DCL command "SHOW DEVICE/FULL <disk name>" to display the values sectors/track, tracks/cylinder and cluster size.

Adabas Block Sizes

If you use the device type independent access method, you should select block sizes for the DATA and WORK container files that are a multiple of the ASSO block size. This minimizes the temporary unused space in the Adabas buffer pool when replacing blocks of different container file types.

With this rule, different combinations of block sizes are possible.

Examples:

ASSO	DATA	WORK
3 K	6 K	6 K
2.5 K	7.5 K	7.5 K
2 K	4 K	8 K

We recommend you to use this rule also if you use the device type dependent access method. When you select the block sizes for use with this method, you should also take into account the number of sectors per track, so that the unused space at the end of the track is not too large.

Example:

If the disk has 62 sectors per track (i.e. the track size is 31K), the following table shows how much unused space there is per track, depending on the block size you choose for the container file.

ASSO/DATA/WORK Blocksize	Adabas blocks per track	unused space at end of track
4 K	7	3 K
8 K	3	7 K
3 K	10	1 K
6 K	5	1 K

Effect of large buffer sizes for PLOG and WORK

If you specify a WORK block size of 8K or less, Adabas will set the PLOG block size to 8K. If you specify a WORK block size larger than 8K, Adabas will set the PLOG block size to 32K.

To ensure that all completed transactions can be re-applied during a database recovery, the PLOG buffer is flushed to the PLOG after each ET command, regardless of whether the PLOG buffer is full or not. Each subsequent ET command causes the current PLOG block to be re-written, as long as the PLOG buffer is not full. A new PLOG block is only started when the PLOG buffer is full. Similarly, to ensure data consistency after an autorestart, the current WORK part 1 block is re-written after each ET command until the WORK part 1 buffer is full.

In general, if you have large PLOG and WORK block sizes, more transactions are required to fill the PLOG buffer and WORK part 1 buffer than with small block sizes. This means that the average size of the I/O transfers is increased, but the total number of I/O transfers due to ET commands is unchanged.

For this reason we recommend you to use a WORK block size of 8K or less if your compressed data records do not exceed 8K, and therefore a PLOG block size of 8K.

Database Auto Expand

If a database becomes full, Adabas is able to auto expand the database containers ASSO and DATA. The prerequsite for this is that the nucleus parameter OPTIONS=AUTO_EXPAND has been specified. The strategy used to allocate new space is as follows:

1. Try to increase the last extent of the container that requires new space. This is only possible if the extent has the same block size as required for the new space in the container.
2. Check whether there is an environment variable for the next container extent. If the environment variable exists, it must contain the file name for the next extent, and the specified location must have enough space available for the new container extent.
3. Check whether the DBnnn.INI files contain entries in the section RESERVED_LOCATIONS. If they do, try to allocate the new container extent in one of the specified locations. Refer to the *Adabas Extended Operations* documentation for further information about the DBnnn.INI files. The file name for the new container extent will be CONTx.nnn, where CONT is ASSO or DATA, x is the extent number and nnn is the 3 digit database ID.
4. Try to allocate the new container extent in the database directory (UNIX: $ADADATADIR/dbnnn, Windows: %ADADATADIR%\dbnnn),. The file name for the new container extent will be CONTx.nnn, where CONT is ASSO or DATA, x is the extent number and nnn is the 3 digit database ID.

Notes:

1. Utilities auto expand the database only in online mode when the nucleus is active. An exception to this is ADABCK, where the database can also be expanded in offline mode.
2. If the auto expand is to be done in a file system, a file with the same name must not already exist. If the auto expand is to be done into a raw section, the raw section must not already contain a container of this type with the same extent number and database ID. It does not matter whether the container extent has only been allocated with ADADEV, or if the container has really been included in a database.
3. If you specify explicit RABNs for space allocations, no auto expand will be performed if the database does not contain all of the requested RABNs.

Index Block Sizes

When Adabas creates index blocks, it allocates blocks with a block size that depends on the descriptor value sizes:

- Large descriptor values >253 bytes are stored in large index blocks with a block size >= 16 KB.
- Smaller descriptor values are stored in small index blocks with a block size < 16 KB.

If you want to store large descriptor value, you must, therefore, define an ASSO container with a large block size for the database.

SORT Data Set Placement

It is recommended that the SORT data set does not reside on the same volume as Associator and DATA. When processing a file which contains more than 100 000 records, the SORT area should be split across two volumes to minimize disk arm movement.

The SORT data set may be omitted when processing only small amounts of data (e.g. when inverting a field in an empty file). The Adabas utility being used then performs an in-core sort.

The SORT data set must be large enough to sort the largest descriptor to be processed. Check the ADACMP or ADAULD log for a list of descriptors, as well as a recommended size of SORT and TEMP for any future data compression or decompression operations.

The ADAINV SUMMARY function also displays the required SORT and LWP size for a memory-resident sort.

Note: If you want to force ADAINV to do a memory-resident sort, do not specify a SORT data set, since otherwise ADAINV might do a file-based sort for the first descriptor, even if the LWP parameter is large enough for a memory-resident sort. This is because ADAINV does not know in advance the size of the descriptor. The subsequent descriptors will always be processed in memory if possible.

TEMP Data Set Placement

It is recommended that the TEMP data set does not reside on the same volume as DATA and SORT.

The TEMP data set is used while

- loading Normal and Main Index;
- updating Upper Index

Although the size of TEMP is closely related to the system performance when loading the Normal/Main Index, successful execution does not depend on a given size. When updating the Upper Index, however, all data required must fit into the TEMP data set.

ADACMP and ADAULD display the recommended TEMP size in the descriptor summary.

The TEMP data set is used for intermediate storage of descriptor values if more than one descriptor is inverted.

Although the size of TEMP is closely related to the performance when loading the Normal/Main Index, successful execution does not depend on a given size or the presence of a TEMP. It is recommended that the TEMP data set should be at least large enough to store the second largest descriptor. If you increase the size of the TEMP data set, the number of passes (i.e. the number of times the DATA area of the processed file is read) can be reduced. The ADAINV/ADAMUP SUMMARY function displays the recommended sizes for the TEMP data set.

Container Files in File System or Raw Device

You can create the Adabas container files either in a file system or on raw devices (UNIX only).

The following points should be considered:

- In general, it is not possible to say whether containers in a file system or containers on raw device are better; this very much depends on the way Adabas is used. A file system has the advantage that it can buffer data, which means that a file system I/O does not necessarily result in a disk I/O, and a file system may optimize the I/O operations. But on the other hand, the file system also means an overhead that is avoided on raw devices. Software AG therefore recommends that you to try both and use the I/O system which delivers the best performance in the given environment.
- Raw devices are limited to 2 terabytes.

Caution: Adabas does not check to see whether a raw device is ≤ 2 terabytes, but if you use larger raw devices, unexpected errors can occur.

- If you want to create containers larger than 2 terabytes, you must create them in a file system.
- If you use containers in a file system and want to have a behaviour similar to that of containers on raw device, it is recommended that you use the ADANUC parameter UNBUFFERED.
- Adabas containers can be created on local disks or on remote storage servers.
- If you use disks on storage servers, the I/O speed may be limited by the speed of the network between your computer where Adabas is running and the storage server; this may decrease the overall performance of Adabas.
- Some file systems that support snapshots of the file system do not overwrite updated blocks, but write a copy to a different location. If there are a lot of updates to the database, the resulting fragmentation of the data may lead to a very poor I/O performance. Software AG recommends that you ask the storage-system vendor if this can happen with his storage system, and what can be done to avoid these problems.
- If the buffer pool is large enough (ADANUC parameter LBP), I/O performance is normally not critical; this is because most logical I/Os do not require a physical I/O. However, the performance of the devices that contain WORK and PLOG (Adabas protection log) is important, since WORK and PLOG contain log information that is required to guarantee database integrity. For this reason, an ET (end of transaction) command can only be completed when the log information is safely stored on the WORK and PLOG devices. Software AG therefore recommends the use of very fast storage devices if you have a high update load; we have seen performance improvements of up to 30% in cases where the normal storage device was replaced by a faster one for WORK and PLOG. Of course, the performance improvement depends very much on the mix of Adabas commands issued and the speed difference between the different storage devices.

4 Temporary Working Space

If the Adabas nucleus or utilities need temporary working space on disk, the TEMP and SORT containers may be used, or space is allocated and subsequently released in temporary working locations. Temporary working locations are directories that should contain sufficient free space for the required working space.

Temporary working space on disks is accessed according to the following rules:

- Only the TEMP container, which must have been defined with the utility ADAFRM, can be used as temporary working space for the intermediate storage of descriptor values in the utilities ADAINV and ADAMUP.

  The search stategy for finding the TEMP containers is as follows:

  1. Check for the environment variables TEMP1, TEMP2 etc. If such an environment variable exists, it must contain the file name of the TEMP container.
  2. Search for the corresponding entries in the DBnnn.INI file. If such an entry exists, it must contain the file name of the TEMP container. Refer to the *Adabas Extended Operation* documentation for further information about the DBnnn.INI files.
  3. Search for the file TEMPx.nnn in the database directory (UNIX: $ADADATADIR/dbnnn, Windows: %ADADATADIR%\dbnnn), where x is the extent number and nnn is the 3 digit database ID.
- The search strategy for finding sort working space in the utilities ADAINV and ADAMUP is as follows:

  1. Check for the environment variables SORT1, SORT2 etc. If such an environment variable exists, it must contain the file name of the SORT container. SORT can have up to 50 extents. Unlike the case with TEMP containers (only files previously created with ADAFRM can be used), files that do not currently exist will be directly created by ADAINV/ADAMUP.
  2. Search for the corresponding entries in the DBnnn.INI file. If such an entry exists, it must contain the file name of the SORT container. Refer to the *Adabas Extended Operation* documentation for further information about the DBnnn.INI files. Unlike the case with TEMP containers

(only files previously created with ADAFRM can be used), files that do not currently exist will be directly created by ADAINV/ADAMUP.

3. Check for the environment variables TEMPLOC1, TEMPLOC2. If the environment variables exist, they must contain the names of directories in which the utilities try to create the SORT containers. The file names will be SORTpid.dbid. The sort files will be deleted when the utility finishes processing.
4. Search for the corresponding entries (TEMPORARY_LOCATION) in the DBnnn.INI file. If such entries exist, they must contain the name of the directories in which the utilities try to create the SORT containers. Refer to the *Adabas Extended Operation* documentation for further information about the DBnnn.INI files. The file names will be SORTpid.dbid.
5. Try to create a file in the database directory (UNIX: $ADADATADIR/dbnnn, Windows: %ADADATADIR%\dbnnn). The file names will be SORTpid.dbid.

   Sort files created by the utilities will be deleted when the utility finishes processing. ADAINV and ADAMUP also include cleanup processing, which deletes sort files that could not be deleted as a result of a utiltity terminating abnormally.

- The search strategy for finding the file names for creating TEMP and SORT containers with ADAFRM corresponds to steps 1 - 3 of the search strategy for ADAINV/ADAMUP.
- Sort and other temporary working space required by ADANUC is created directly by ADANUC itself according to the following search strategy:

  1. Check for the environment variables TEMPLOC1, TEMPLOC2. If the environment variables exist, they must contain the names of directories or raw sections in which ADANUC tries to create the temporary containers.
  2. Search for the corresponding entries (TEMPORARY_LOCATION) in the DBnnn.INI file. If such entries exist, they must contain the name of directories where ADANUC tries to create the temporary containers. Refer to the *Adabas Extended Operation* documentation for further information about the DBnnn.INI files.
  3. Try to create the files in the database directory (UNIX: $ADADATADIR/dbnnn, Windows: %ADADATADIR%\dbnnn) - this applies only if ADANUC is not started in read-only mode.

  The file names will beNUCSRTx.dbid and NUCTMPx.dbid. The files will be deleted again during the shutdown processing of ADANUC.

Note: Do not specify raw sections for temporary locations for utilities (UNIX platforms only). In order to make the created file name unique, the PID is added to the file name for the created files in the temporary locations, and this is not possible in raw sections. If you want to use raw sections for temporary working space, create the TEMP and SORT containers in a raw section with ADAFRM and assign them to the SORTn and or TEMPn environment variables. You must take care to ensure that the SORT and TEMP containers are used only by one utility at a time.

5 FDT Record Structure

This chapter describes the syntax and use of the data definitions to define the layout of files in the database. This input has to be contained in the sequential file FDUFDT that is input to the utility ADAFDU.

The data definitions are used to create the field definition table (FDT) for the file. This table is used by Adabas while executing Adabas commands to determine the logical structure and characteristics of any given field (or group) in the file.

Data Definition Syntax

A separate data-definition statement is required for each field or group to be defined.

The syntax used in constructing data definition entries is:

```
level-number, name [,standard_length, standard_format] [(,definition_option)...]
```

`level number' and `name' are required. Any number of spaces may be inserted between definition entries in a line. All text behind a semicolon is treated as comment, and a line that starts with a semicolon is treated as a comment line. Any number of empty lines is allowed.

Level Number

The level number is a one- or two-digit number in the range 01...07 used in conjunction with field grouping. Leading zeros are optional. Fields may be defined at levels in the range 01...07, where any field with a level number of 02 or greater is considered to be a member of the group on the next lowest level.

Groups may be defined on levels in the range 01...06 and may contain other groups. Level numbers may not be skipped when assigning the level numbers for a group.

The definition of a group enables the user to reference a series of fields (may also be only one field) by using the group name. This is a convenient and efficient method of referencing a series of consecutive fields.

Example

```
01,GA     ; Group
02,A1,...    ; Elementary or Multiple Value field
02,A2,...    ; Elementary or Multiple Value field
01,GB     ; Group
02,B1,...    ; Elementary or Multiple Value field
02,GC     ; Group
03,C1,...    ; Elementary or Multiple Value field
03,C2,...    ; Elementary or Multiple Value field
```

Fields A1 and A2 are in group GA. Field B1 and group GC (consisting of fields C1 and C2) are in group GB. The periods (...) denote further specifications.

Name

The name to be assigned to the field or group.

The name must be two characters in length. The first character must be alphabetic and the second character alphabetic or numeric; upper case and lower case characters are allowed. No special characters are permitted. A maximum of 3214 fields can be defined in a single Adabas record.

The values E0 through E9 are reserved as edit masks and may not be used (see Calling Adabas in the Command Reference Manual for further information about edit masks).

Names must be unique within a file. Names which are English prepositions or articles such as AN, AT, BY, IF, IN, OF, ON etc. should not be used because of possible conflict with syntactical terms used by NATURAL.

```
Valid Names       Invalid Names
-----------       -------------

    AA                A  (not 2 characters)
    e3                E3 (edit mask)
    S3                F* (special character)
    wm                3M (first character not alphabetic)
```

Standard Length

The standard length of the field (expressed in bytes). Standard length is used to define the standard (default) length to be used by Adabas during command processing. The standard length specified is entered in the field definition table (FDT) and used when the field is read/updated, unless the user specifies a length override.

The maximum field lengths which may be specified are:

Format	Maximum Length
ALPHANUMERIC	LA, L4/LB option: 16381 bytes, if no LOB file is defined or if the field is a descriptor or a parent of a derived descriptor. Otherwise 65533 for an LA field and 2147483543 for an L4/LB option, else: 253 bytes
BINARY	126 bytes
FIXED POINT	8 bytes (1,2,4,8 bytes only)
FLOATING	8 bytes (4, 8 bytes only)
PACKED DECIMAL	15 bytes
UNPACKED DECIMAL	29 bytes

Format	Maximum Length
UNICODE	LA, L4 option: 16381 bytes in UTF-8 encoding else: 253 bytes in UTF-8 encoding (see note below)

Note: The length of a Unicode field depends on the encoding used. Internally, Adabas uses UTF-8 encoding to store Unicode fields, but the Adabas user can use other encodings to access Unicode fields, and there is no fixed maximum size for a field in this encoding.

Standard length may not be specified with a group name.

Standard length does not limit the size of any given field value (unless the FI option is used). The user may issue a READ or UPDATE command in which a length greater than the standard length is specified.

If standard length is omitted for a field, the field is assumed to be a variable- length field. Variable-length fields have no standard (default) length. If a variable-length field is referenced without a length override during an Adabas command, the value of the field will be returned preceded by a one-byte field which contains the length of the value (including length byte). The user must give this length byte when the field is updated.

Standard Format

The standard format of the field (expressed as a one-character code):

Code	Format
A	Alphanumeric (left-justified)
B	Binary (right-justified, unsigned)
F	Fixed point (right-justified, signed)
G	Floating (floating, double precision)
P	Packed decimal (right-justified, signed)
U	Unpacked decimal (right-justified, signed)
W	Unicode (see note below)

Note: The field is stored internally in UTF-8 encoding, but when you access the field you can specify a different encoding, to or from which the value is converted.

The standard format is used to define the standard (default) format to be used by Adabas during command processing. The standard format specified is entered in the field definition table and is used when the field is read/updated, unless the user specifies a format override.

Standard format must be specified for a field. It may not be specified with a group name. A group has no default format. When a group is referenced, the fields within the group are always returned, or must be provided, according to the standard format of each individual field.

Definition Options

Definition options are specified by two-character codes as described below. These codes may be specified in any order, separated by a comma, as the last entries of a data definition statement.

Descriptor (DE)

DE indicates that the field is to be a descriptor. Entries will be made in the Associator inverted list for the field, enabling this field to be used in a search expression, as a sort key in a FIND command or to control logical sequential reading.

A maximum of 256 descriptors (including phonetic descriptors, subdescriptors, superdescriptors and hyperdescriptors) may be specified for a file.

The descriptor option should be used judiciously, particularly if the file is large and the field being considered as a descriptor is updated frequently.

Date/Time (DT)

There are various ways in which date/time values can be stored in the database, e.g.:

- Timestamps in the format YYYYMMDDhhmmss
- Natural date/time fields
- UNIX time_t

The date/time edit mask specified with the DT option defines which date/time format is used to store the date/time values internally.

The syntax for the field option DT in a field definition is

```
DT=date_time_edit_mask
```

where date_time_edit_mask is

```
E(date_time_edit_mask_name)
```

The following date / time edit masks are supported:

Date_time_edit_mask	Description	Value 0 is
E(DATE)	Date: YYYYMMDD	Invalid date – unknown
E(TIME)	Time: HHIISS	00:00:00
E(DATETIME)	Date and time: YYYYMMDDHHIISS	Invalid date – unknown
E(TIMESTAMP)	Numeric timestamp with microsecond precision: YYYYMMDDHHIISS6	Invalid date – unknown
E(NATTIME)	Natural T format (tenths of seconds since 0000-01-02)	0000-01-02:00:00:00.0 before year 1 – unknown
E(NATDATE)	Natural D format (days since year 0000)	0000-01-02 before year 1 – unknown
E(UNIXTIME)	UNIX time_t type (seconds since 1970), always UTC (Coordinated Universal Time) based	1970-01-01:00:00:00
E(XTIMESTAMP)	UNIX timestamp with microseconds since 1970 (UNIXTIME * 1000000) + microseconds, always UTC-based	1970-01-01:00:00:00.000000

The DT option is only allowed with the formats B, F, P, U. The length specified for a field with the DT option must be large enough to store the date/time values. The following table shows the required minimum lengths and if the field option TZ (local time zone) is allowed (for more information see description of TZ):

Date_time_edit_mask	Required minimum field lengths for format				TZ Option Allowed
	B	F	P	U	
E(DATE)	4	4	5	8	no
E(TIME)	3	4	4	6	no
E(DATETIME)	6	8	8	14	yes
E(TIMESTAMP)	-	-	11	20	yes
E(NATTIME)	5	8	7	12	yes
E(NATDATE)	3	4	4	6	no
E(UNIXTIME)	4	4	6	10	yes
E(XTIMESTAMP)	7	8	9	16	yes

Notes:

- The formats B and F are not allowed with E(TIMESTAMP).
- If you use date/time edit masks, date/time values between 0001-01-01:00:00:00.000000 and 9999:12-31:23:59:59.999999 are allowed. The value 0 is allowed also for those date/time edit masks where 0 is not a valid date/time value. In these cases, the meaning of value 0 is “unknown”. If the value is specified for an NC field, the significance indicator is set to -1, independent of a significance indicator provided in the format/record buffer. If you convert a date/time edit mask with value 0, and 0 represents an unknown date to a field with another date/time edit mask, the result is always 0. If the target field is an NC field, the significance indicator is set to -1.

- Dates before 1592, when the Gregorian calendar was introduced, are handled as if the Gregorian calendar was also valid before the dates in question:
  - You can enter dates that did not exist historically;
  - Dates that existed historically, but which are not defined in the proleptic Gregorian calendar are rejected for NATDATE, NATTIME, UNIXTIME and XTIMESTAMP;
  - If you compute time intervals, you may get results that are not equal to the historical time intervals.
- For DATE, DATETIME and TIMESTAMP, it is possible to specify dates that existed historically according to the Julian calendar, but which do not exist in the proleptic Gregorian calendar; it is the responsibility of user which semantics he assigns to such date/time fields. However, if you try to convert such a date to another date/time edit mask, you get an error (Adabas response code 55).
- Depending on the format/length used for UNIXTIME or XTIMESTAMP fields, it is possible that only a subset of the range between years 0 and 9999 is supported:
  - If you use the format B for UNIXTIME or XTIMESTAMP fields, it is not possible to store date/time values before 1970 - this would require negative values that are not supported with the format B.
  - If you use the format F with a length of 4 for UNIXTIME, you will not be able to store date/time values after January 19, 2038.
  - If you use the format B with a length of 4 for UNIXTIME, you can specify date/time values until the year 2106.
  - If you use the format U with a length of 10, the maximum date for UNIXTIME is in the 23rd century.
- Fields with date/time edit masks should not be used to store time intervals.

Adding the DT Option to Fields in exisiting Files

It is possible to add the DT option to fields in files that already exist. In order to guarantee compatibility with existing applications, fields with defined with the DT option (but without the TZ option) are handled as follows:

- Adabas does not check whether if all values stored before adding the DT option are correct date/time values - this is the resposibility of the user. It is up to the user to care for the integrity of the file
- If you attempt to read the field with a date/time edit mask and the field value is not a valid date/time value, you will get an Adabas response code 55. If you read the field without a date/time edit mask, Adabas does not check whether the value is a correct date/time value.
- If you don't specify a date/time edit mask for the field in the format buffer for an add/update command, the field is processed as if the field was defined without the DT option - no checks are made for correct date/time values. In order to ensure that the field contains correct date/time values, it is recommended to use date/time edit masks in the format buffer for all updates made

to date/time fields - in this case, then invalid date/time values are rejected with an Adabas response code 55.

Fixed Storage (FI)

FI indicates that the field is to occupy a fixed amount of storage and is not to be compressed.

In the Data Storage, the field value is stored without an internal length byte.

The FI option is recommended for fields with a length of 1 or 2 bytes which have a low probability of containing a null value, as well as for fields containing non-compressible values.

The FI option is not recommended for fields defined as multiple-value fields or for fields in a periodic group at the end of a record. Any null values for such fields will not be suppressed (or compressed), which may result in considerable waste of disk storage and increased processing times.

Example

```
                    Without            With
                    FI option          FI option
                    ---------          ---------

Definition          01,AA,3,P          01,AA,3,P,FI

User Data           33104C             33104C

Internal            0433104C           33104C
Representation      (4 bytes)          (3 bytes)

User Data           00003C             00003C

Internal            023C               00003C
Representation      (2 bytes)          (3 bytes)
```

Restrictions on FI usage:

- The FI, NC and NU options are mutually exclusive;
- The FI option must not be specified for variable-length fields (standard length omitted);
- A field defined with the FI option cannot be updated with a value which exceeds the standard length of the field.

High-Order First (HF)

The Adabas binary field format B is used by applications in two ways, either for unsigned integer values or for bit strings with arbitrary bit combinations and length. While in the first case, the values are expected to be ordered according to the hardware architecture and to be swapped if exchanged between different integer architectures, in the second case, the values are always interpreted as high-order-first (Big Endian) values; this means that the values are not swapped when exchanged between different integer architectures.

In order to enable both kinds of usage, the high-order first option (HF) was introduced for binary fields:

- If a binary field is defined without the HF option, the values are interpreted as unsigned integers according to the byte order defined on the current hardware.
- If a binary field is defined with the HF option, the values are always interpreted as high-order-first values, also on low-order-first platforms.

Note: Natural expects fields with the Natural format B to always be binary fields defined with HF option. If you use binary fields without the option HF in Natural, you will get processing errors:
- If you access these fields from a database on a machine with a different integer architecture, the values will be swapped.
- If the fields are descriptors and are stored on a machine with low-order-first architecture, the sort sequence will not be as expected.

If a B field that is defined with the HF option is a part of a superdescriptor and the format of the resulting superdescriptor is not alpha, then the HF option is also applied to the superdescriptor. This means that the superdescriptor values are stored in the high-order first format.

Example:

The following fields are defined in the FDT:

```
1,B1,4,B
1,B2,4,B,HF
```

The following format buffer is defined:

```
FB="B1,B2"
```

and the following array is used as a record buffer:

```
unsigned char RB[8];
```

The record buffer on a high-order first machine is now filled with the following commands:

```
RB[0] = 1;  /* B1 = 0x01020304 high-order first*/
RB[1] = 2;
RB[2] = 3;
RB[3] = 4;

RB[4] = 1;  /* B2 = { 1, 2, 3, 4 } */
RB[5] = 2;
RB[6] = 3;
RB[7] = 4;
```

Reading the values from a low-order first machine returns the following values:

```
RB[0] = 4;  /* B1 = 0x01020304 low-order first*/
RB[1] = 3;
RB[2] = 2;
RB[3] = 1;

RB[4] = 1;  /* B2 = { 1, 2, 3, 4 } */
RB[5] = 2;
RB[6] = 3;
RB[7] = 4;
```

Long Alpha (LB/L4, LA)

The LB/L4 (long alphanumeric - 4 bytes length) or LA (long alphanumeric 2 bytes length) option can be specified for alphanumeric and Unicode fields. LB and L4 are synonyms. Only one of the LB/L4 or LA options can be specified for a given field. A field defined with the LB/L4 or LA option can contain a value that is up to 16,381 bytes long

- if the field is defined as descriptor;
- or if it is a parent field for a derived descriptor;
- or if no LOB file is associated to the file
- or if the field is a Unicode field.

Note: In these cases, the field value is always stored in the primary record. If you define such a field, you should consider that the primary record must fit into a data block, which can have a size of up to 32 KB. You should only define such fields if it will not result in a record overflow.

Otherwise the value can be up to 65533 bytes long for LA fields, and up to 2147483543 bytes for LB/L4 fields.

If a LOB file is associated with the file, or if the field is not a descriptor or a parent field of a derived descriptor and the value length is > 253, the field value is stored in the LOB file, and a LOB reference is included in the base record. Otherwise the field is compressed the same way as a field without the LB/L4 or LA option. The maximum length that a field with LA option can actually have is limited by the block size of the block in which the compressed record is stored - the compressed record must fit into one block.

When a field with LA option is updated or read with variable length, its value is either specified or returned in the record buffer, preceded by an inclusive two-byte length value (field length, plus two).

When a field with L4 option is updated or read with variable length, its value is either specified or returned in the record buffer, preceded by an inclusive 4-byte length value (field length, plus 4).

A field with the L4 or LA option

- can also have the NU, NC/NN, or MU option;
- can be a member of a PE group;
- cannot have the FI option;
- can be a descriptor field, but in this case only values with a maximum length of 1144 (exclusive field length) can be stored if the field does not have the TR option. If the descriptor field has the TR option, values larger than 1144 bytes are possible, but the descriptor value in the index is truncated to 1144 bytes.

Example of L4/LA usage

Option	Definition	User data (variable length) (high order first)	User data (variable length) (low order first)
Without L4 or LA	01,BA,0,A	"\x06HELLO"	"\x06HELLO"
With L4	01,BA,0,A,L4	"\x00\x00\x00\x09HELLO" "\x00\x00\x07\xD4" (2000 data bytes)	"\x09\x00\x00\x00HELLO" "\xD4\x07\x00\x00" (2000 data bytes)
With LA	01,BA,0,A,LA	"\x00\x09HELLO" "\x07\xD2" (2000 data bytes)	"\x09\x00HELLO" "\xD2\x07" (2000 data bytes)

Multiple-Value Field (MU)

MU indicates that the field may consist of 0, 1 or more than one value.

The values are stored internally according to the other options specified for the field. For an NU option field, trailing empty values are suppressed. The MU and NC options are mutually exclusive.

The syntax MU(n), as used in the utility ADACMP, is accepted but the occurrence count is ignored.

Example (MU with NU)

```
Definition:  01,AA,5,A,MU,NU

Original content after file loading:

       3   L  value A   L  value B   L  value C

     count    field AA1    field AA2    field AA3
```

L means length of the following value, including the L byte.

After update of value B to empty value:

```
       2   L  value A  L   value C

     count    field AA1    field AA2
```

AA count = 2.

Example (MU without NU)

```
Definition:  01,AA,5,A,MU

Original content after file loading:

       3   L  value A  L   value B   L   value C

     count    field AA1    field AA2     field AA3
```

After update of value B to null value:

```
      3   L  value A  L   null value  L  value C

    count    field AA1    field AA2    field AA3
```

AA count = 3.

No Blank Compression (NB)

The NB option indicates that trailing blanks are not suppressed when a value is stored; values are always stored in the database with the same length as specified in the record buffer. A string which has a value that corresponds to the beginning of another string will always be considered as having a value less than the other string. This has the following consequences for the order of values:

Without NB option

```
"xxx\x00\x00" < "xxx\x00" < "xxx" = "xxx " = "xxx  " < "xxx0"
```

With NB option

```
"xxx" < "xxx\x00" < "xxx\x00\x00" < "xxx " < "xxx  " < "xxx0"
```

The NB option is not allowed together with the FI option. The NB option is only allowed for fields with the format A or W.

If a value defined with the NB option is read with a fixed length that is larger than the value length, the value is filled with trailing blanks, like the value for a field without the NB option. However, if you perform an update with the same format and record buffer, the value is modified in the database – the trailing blanks are appended to the value.

SQL Null-Value Representation (NC)

The NC option indicates that the field can represent NULL values that are used by SQL. If this option is used, the field that contains an empty value can be in one of two states:

- not present (NULL)
- empty (blank)

A special format-buffer element (the S element) indicates whether the field is empty or not present. Please refer to the section Format Buffer Syntax in the Command Reference Manual for further information.

The FI, NU and NC options are mutually exclusive. The NC option is not permitted with a multiple-value field, and must not be specified for a member of a periodic group.

Example

```
Definition: 01,AA,2,B,NC
```

```
    Value  Blank  NULL

User S element 0   0   -1

User data  0005   0000   0000

Internal   0205   01   C1
representation
```

Not Null Option (NN)

A field that is defined with the NN option must always be assigned a value during an update or add. A value or blank must be provided in each data record, otherwise Adabas returns a response code. This option may only be specified in conjunction with the NC option.

Example

```
Definition:  01,AA,2,B,NC    01,AA,2,B,NC,NN
```

```
User S element -1      -1

User data  0000      0000

Internal   C1      not permitted
representation
```

Null Value Suppression (NU)

NU indicates that null values for the field will be suppressed.

Null value suppression results in the internal representation of a null value by a one-byte empty field indicator. The null value is not stored.

A series of consecutive fields, each of which contains a null value and for which the NU option is defined, is represented internally by a one-byte empty field indicator which contains the number of successive fields containing a null value. Hence, fields defined with the NU option should be defined in consecutive order whenever possible.

If the NU option is specified for a descriptor, a null value for the descriptor is not stored in the inverted list. Therefore, a FIND command in which a null value for this descriptor is used will always result in no records found, even though there may be records containing a null value in Data Storage.

If a descriptor defined with the NU option is used to control a logical sequence in a READ LOGICAL SEQUENCE command, those records which contain a null value for the descriptor will not be read. If the descriptor has both the NU and the UQ options, null values could be stored multiple times without there being a uniqueness violation.

The FI, NC and NU options are mutually exclusive.

Normal compression (NU or FI not specified) results in the representation of a null value by 1 byte.

Example (compression)

```
                Normal          With FI       With NU
                Compression     Option        Option
                -----------     -------       -------

Definition      01,AA,2,B       01,AA,2,B,FI  01,AA,2,B,NU

User data       0000            0000          0000

Internal        01              0000          C1
Representation  (1 byte)       (2 bytes)     (1 byte)
                                              C1 indicates
                                              1 empty field
                                              follows
```

No Value Conversion (NV)

A field that is defined with the NV option will not be converted if an UPDATE or READ command is received from a machine with a different architecture.

The NV option cannot be specified for Unicode fields.

Example

```
Definition:       01,AA,2,A           01,AA,2,A,NV

EBCDIC data has   convert value
to be stored in   of AA from EBCDIC   no conversion
a database on an  to ASCII
ASCII machine
```

Periodic Group (PE)

PE indicates that a periodic group is to be defined.

A periodic group may consist of one or more fields and may occur zero times, once or more than once within a given record.

The periodic group is defined at the 01 level. All of the fields to be included in the periodic group must follow immediately and must be defined at level 02 or higher (in increments of 1 to a maximum of 7). The next 01 level definition indicates the end of the periodic group.

PE may only be specified with a group name. Length and format parameters may not be specified with the group name. A periodic group may contain descriptors and/or multiple-value fields and other groups but may not contain another periodic group.

Example

```
01,GA,PE              ; PERIODIC GROUP GA
  02,A1,6,A,NU
  02,A2,2,B,NU
  02,A3,4,P,NU
01,GB,PE              ; PERIODIC GROUP GB
  02,B1,4,A,DE,NU
  02,B2,5,A,MU,NU     ; MU fields in PE groups
                      ; are permitted.
  02,B3               ; Grouping of fields within
    03,B4,20,A,NU     ; PE groups is permitted.
    03,B5,7,U,NU
```

```
01,XA,PE
  02,X1,3,A,NU
  02,X2,4,U,NU
  02,YA,PE            ; Invalid. Nested periodic group not permitted.
         ^
%ADAFDU-E-PGL1  periodic group may only be defined at level 1
```

The NU option is recommended for fields within a periodic group. This permits maximum compression and results in less processing time during read/update of the fields.

System Generated (SY)

The values for system generated fields are automatically generated by Adabas - values specified in the record buffer in an update or store command, are ignored. A system generated field must not be a field in a periodic group. A system generated field with the CR option must not be a multiple-value field.

System generated fields are defined with the following syntax:

```
SY = keyword [,CR]
```

where keyword can take the following values:

TIME
Creation or last update timestamp. The field must be defined with the DT option. The values are stored as UTC values. If you want to access the values as local time values, the field must be defined with the TZ option.

Example:

```
1,CR,14,U,DE,DT=E(DATETIME),TZ,SY=TIME,CR  ; Creation timestamp
```

SESSIONID
The Adabas session ID of the Adabas user session in which the record was created. The field must be defined with the options A,NV. The recommended field length is 28. If a smaller length is provided in the field definition, the value is truncated. The layout is shown below:

Bytes	Meaning
unsigned char s_node[8]:	Adabas client node name
unsigned char s_user[8]:	Adabas client user ID
unsigned int s_pid[4]:	Process identification
unsigned char s_timestamp[8]:	Session timestamp: microseconds since 1970, as binary value

Notes:

1. The Adabas session ID is a binary string which identifies an Adabas session; there is no conversion between platforms. At a given time, the Adabas session ID is unique, but later on Adabas session IDs can be reused. On mainframes the layout is different than on open systems.
2. You can change the Adabas session ID with the function lnk_set_adabas_id. This means that there is no guarantee that the components of this Adabas session ID really contain information on the user.
3. The issues mentioned above have to be taken into account if applications want to access the components of SESSIONID.

4. The session timestamp is defined as unsigned char[8] because of alignment reasons, but it contains a binary value.
5. The session timestamp on open systems platforms is 0 if an Adabas version < 6.2. SP2 or a Net-Work version 7.3 is used, this is because earlier versions still used a 20-byte session ID without a timestamp.
6. These rules for the layout of the Adabas session ID only apply to open systems platforms; on mainframes there is also a 28-byte Adabas session ID, but the components are different. Please refer to the mainframe documentation for details.

Example (open systems):

```
1,CA,16,A,NV,SY=SESSIONID   ; Node ID and client user ID of last update
UN=CA(9,16)      ; Subdescriptor for user ID of last update
```

SESSIONUSER

The login ID of the Adabas user session in which the record was created or updated. The field must be defined with format A. The recommended field length is 8. The value of a SESSIONUSER field is bytes 9 - 16 of a SESSIONID field.

Note: If you have also defined a SESSIONID field, you can define a subdescriptor of this field instead of the SESSIONUSER field - see the example for SESSIONID.

Example:

```
1,CU,8,A,DE,SY=SESSIONUSER,CR  ; Login ID of creator
```

OPUSER

The user ID specified in Additions 1 of the OP command for the Adabas session in which the record was created. The field must have format A and length 8.

Example:

```
1,CO,8,A,DE,SY=OPUSER   ; User ID specified in OP command for last update
```

System Generated Fields with Option CR (Creation)

The values for system generated fields with the option CR are automatically generated by Adabas when a record is created. They are not changed by further update operations.

System Generated Fields without Option CR and without Option MU

The values for system generated fields without the option CR and without the option MU are automatically generated by Adabas when a record is created. The values are updated during each following update operation.

System Generated Fields without Option CR and with Option MU

System generated fields with the option MU can have up to SYFMAX (file parameter, for further information refer to the documentation of the utility ADAFDU) values.

When a record is created, the first value of the MU field is generated.

When a record is updated, a new value is generated and added before the first existing value to the MU field.

Afterwards, if values with an MU index > SYFMAX exist, these values are removed, e.g. assume the field name for the SY fields is SY, then for indices 1 < i <= SYFMAX, the new value of SY(i) is the old value of SY(i-1).

System Generated Fields and the Utility ADACMP

If you use ADACMP in order to perform a bulk load of external data with ADAMUP afterwards, these external data may either already contain the values for the system generated fields or not. Therefore, you can specify via the ADACMP parameter SYFINPUT how to handle system generated fields in ADACMP:

- If you specify SYFINPUT=SYSTEM, ADACMP will create the values for the system generated fields as if inserted by the ADACMP process in the database;
- If you specify SYFINPUT=USER, the system generated fields are handled by ADACMP as fields without the SY option.

For further information refer to the documentation of the utility ADACMP.

Index Truncation (TR)

The TR option must be specified with the L4/LA option and the DE option.

The maximum length of a descriptor value is 1144 bytes. If the descriptor is not defined with the TR option, all update operations that insert a descriptor value larger than 1144 bytes are rejected. If the TR option is specified, these values can be inserted in the database, but the descriptor value will be truncated in the index. The consequence of this is that search operations no longer return the exact result if there is more than 1 record with the same descriptor value truncated to 1144 bytes in the index. If this happens, a warning will be issued. The detailed behaviour of descriptors defined with the TR option is as follows:

- If a descriptor value which is larger than 1144 bytes is inserted in the database, the value is truncated in the index, and you receive a response code 2.
- If you perform a search operation for which the result may be not exact as a consequence of truncation, you receive a response code 2.
- If you sort by a descriptor that is defined with the TR option, and there is more than one record with the same, possibly truncated descriptor value in the index, you receive a response code 2.
- A read logical operation (L3/6/9) receives a response code 2 if there is more than one record with the same, possibly truncated descriptor value in the index.
- A check truncation option is available for the S1 command: if you specify this option, the search buffer should contain the name of a descriptor defined with the TR option, and the value buffer should contain the value to be checked. You receive a response code 0 if the value is not truncated, and a response code 2 if the value is truncated. A search operation in the database is not performed.

If you specify the TR option together with the UQ option, a uniqueness error will occur if you store two different descriptor values which are identical following truncation to 1144 bytes.

Local Time Zone (TZ)

If this option is specified, the output field values are expected to be used in local time (or according to a user-defined time zone), and internally the values are stored in UTC. This option is only allowed together with option DT and the date/time edit mask names DATETIME, TIMESTAMP and NATTIME, UNIXTIME and XTIMESTAMP.

Notes:

- TZ is not allowed with DATE, TIME and NATDATE, since time zones are only relevant if both date and time information is available.
- By definition, UNIXTIME and XTIMESTAMP are based on UTC; the standard conversion routines available for these values include the time zone handling. However, you must define UNIXTIME and XTIMESTAMP fields with the option TZ if you want to convert them to or from local time with one of the other date/time edit masks. If the field is defined without the option TZ, it is assumed that the time zone of the external value is UTC.
- It is up to the user if he wants to use a field defined with the date/time edit masks DATETIME, TIMESTAMP or NATTIME and without the TZ option to store UTC time values or local time values. However, the following must be taken into consideration:
  - If you convert such a field to or from the date/time edit mask UNIXTIME or XTIMESTAMP, Adabas assumes that the internal values contain UTC time values.
  - If you use such a field to store local time values, it is not possible to uniquely specify the hour that occurs twice, when the daylight saving time is switched back to standard time.

  If either of these points would be a problem, you should define the field with the option TZ.

- If a field in an existing file contains UTC values, and you want to add the DT and TZ option with one of the date/time edit masks DATETIME, TIMESTAMP or NATTIME, you can do so by adding the new options with ADADBM. If the file contains local time values, you must unload and decompress the original file. Then you can compress and reload the file with the new options.
- If you access fields with the TZ option, and don't specify a date/time edit mask in the format buffer, the fields are processed in the same way as if the date/time edit masks in their field definitions were specified in the format buffer.

Unique Descriptor (UQ)

UQ indicates that the field is to be a unique descriptor. A unique descriptor must contain a different value for each record in the file. However, a multiple-value field may contain the same value several times in one record.

The UQ option must be specified together with the DE option. It is possible to specify the UQ option for more than one field in a file.

Subdescriptor

A subdescriptor is a descriptor derived from a portion of an elementary field. The elementary field may or may not be a descriptor itself. A subdescriptor may also be defined for a multiple-value field or a field in a periodic group, but may not be defined for a particular value of a multiple-value field or for a particular occurrence of a periodic group.

Subdescriptors must be defined after the last field definition.

A subdescriptor has the same format as the field from which it is derived, except fixed point and floating point, which become binary, and Unicode, which becomes alphanumeric.

A subdescriptor which is derived from a packed value has the sign of the source value appended.

Subdescriptor Definition Syntax

```
name [,UQ]= field-name (from, to)
```

name
The name of the subdescriptor. The naming conventions for a subdescriptor are identical to those defined for Adabas names.

field-name
The name of the source field from which the subdescriptor element is to be derived.

The source field may be:

- an elementary field;

- a multiple-value field;
- in a periodic group;
- a descriptor or non-descriptor.

The source field must *NOT* be:

- a particular multiple-value field value;
- a particular periodic group occurrence;
- another superdescriptor, subdescriptor, or phonetic descriptor;

A subdescriptor has the NU/NC option when the source field is defined with the NU/NC option. Therefore, when the source field is empty, the subdescriptor is empty and is not entered in the inverted list.

from
Indicates the relative byte position within the source field where the subdescriptor definition is to begin.

to
Indicates the relative byte position within the source field where the subdescriptor definition is to end.

`from' and `to' are counted from left to right, beginning with 1, for alphanumeric fields and Unicode fields.

`from' and `to' are counted from right to left, beginning with 1, for unpacked and packed fields. If the source field is defined with P format, the sign of the resulting subdescriptor value is taken from the four low-order bits of the low-order byte (byte 1).

`from' and `to' are counted from low order to high order, beginning with 1, for binary, fixed point and floating point fields.

`to' must be less than or equal to 253.

UQ
A subdescriptor can be defined as a unique descriptor.

Subdescriptor Standard Length and Format

A subdescriptor's standard length is defined by the length of the sub-elements and is used by Adabas while processing search commands. For example, a search buffer containing only a sub-descriptor name, without length override, will use this standard length.

Subdescriptors with Unicode Parent Fields

Subdescriptor components that are derived from W fields are created from the internal encoding of the W field (UTF-8). A conversion to or from the user encoding defined for the user session is not performed.

Examples of Subdescriptor Definitions

Example

```
Source Field Definition:      01,AR,10,A,NU

Subdescriptor Definition:     SB = AR(1,5)
```

The values for subdescriptor SB are derived from the first 5 bytes (counting from left to right) of all the values for the source field AR.

```
AR values                     SB values
---------                     ---------

DAVENPORT                     DAVEN
FORD                          FORD
WILSON                        WILSO
```

Example

```
Source Field Definition:      02,PF,6,P

Subdescriptor Definition:     PS = PF(4,6)
```

The values for subdescriptor PS are derived from bytes 4 to 6 (counting from right to left) of all the values for the source field PF.

```
PF values                     PS values
---------                     ---------
(shown in hex)                (shown in hex)

00243182655C                    02431C
00000000186C                        0C*
78426281448D                  0784262D
```

* If the NU option had been specified for PF, no value would have been created for PS for this value.

Example

```
Source Field Definition:      02,PF,6,P

Subdescriptor Definition:     PT = PF(1,3)
```

The values for PT are derived from bytes 1 to 3 (counting from right to left) of all the values for PF.

```
PF values                     PT values
---------                     ---------
(shown in hex)                (shown in hex)

00243182655C                  82655C
00000000186C                    186C
78426281448D                  81448D
```

Superdescriptor

A superdescriptor is a descriptor derived from several fields, portions of fields, or a combination thereof. Each source field (or portion of a field) used to define a superdescriptor is termed an element. A superdescriptor may be defined using from 2 to 20 elements.

Superdescriptors must be defined after the last field definition (before and/or after subdescriptor definitions).

All field formats are accepted as part of a superdescriptor.

Notes:

1. Only the first 253 bytes of the parent fields can be specified.
2. Mainframe Adabas databases do not allow fields in floating point format (format G) to be used as superdescriptor parent fields; open systems Adabas databases do allow fields in floating point format to be used as superdescriptor parent fields.

Superdescriptor Definition Syntax

```
name [,format] [,PF] [,UQ] = field-name (from, to[, encoding]),
                                          field-name (from, to[, encoding])
                                            [[,field-name (from, to[, ↩
encoding])]...]
```

name

The name of the superdescriptor. The naming conventions for superdescriptors are identical to those for Adabas names.

format

The format may only be specified if

- all parent fields have unpacked format. Then A (alphanumeric), B (binary) or U (unpacked) can be specified. For reasons of compatibility with earlier versions of Adabas, the default is B, but it is strongly recommended to always specify either A or U, as the superdescriptor behaviour on low-order-first platforms may lead to strange results.
- • At least one parent field has W format. Then A or W may be specified. The default is A.

PF

Specifying this option ensures compatibility with Adabas databases on mainframe systems if the superdescriptor includes a packed field. On a mainframe database, the sign half-byte of a packed value is 0x0F, whereas under UNIX/Windows it is 0x0C. Using the PF option means that packed positive signs are stored as 0x0F within the superdescriptor.

Note: Although in the index the sign half-byte is 0x0F, you don't get the 0x0F if you specify the superdescriptor in the format buffer for a read command - the sign half-byte is converted to 0x0C. This also means that the sort sequence of the values in the index may be different from the sort sequence that you get if you perform an alphanumeric comparison of the superdescriptor values you have read. If you want to read a range of descriptor values, it is recommended that you specify the end criterion in the search buffer for the L3 or L9 command, and *not* to check the read descriptor values in order to find out if you have met the end criterion.

UQ

A superdescriptor can be defined as a unique descriptor.

field-name

The name of the source field from which a superdescriptor element is to be derived.

The source field may be:

- an elementary field;
- a multiple-value field but only one per superdescriptor;
- any elementary field in a periodic group;
- a descriptor or non-descriptor.

The source field must *NOT* be:

- a particular multiple-value field value;
- a particular periodic group occurrence;
- another superdescriptor, subdescriptor, hyperdescriptor or phonetic descriptor.

A superdescriptor has the NU/NC option when one or more source field is defined with the NU/NC option. Therefore, when one or more of the elements is empty, the superdescriptor is empty and is not entered in the inverted list.

from

Indicates the relative byte position within the source field where the superdescriptor element is to begin.

to

Indicates the relative byte position within the source field where the superdescriptor element is to end.

`from' and `to' are counted from left to right, beginning with 1, for alphanumeric fields and Unicode fields.

`from' and `to' are counted from right to left, beginning with 1, for unpacked and packed fields.

`from' and `to' are counted from low order to high order, beginning with 1, for binary, fixed point and floating point fields.

`to' must be less than or equal to 253.

`from' must be less than or equal to `to'. The total length of any superdescriptor value may not exceed 1144 bytes in the case of alphanumeric, 126 bytes in the case of binary and 29 bytes in the case of unpacked.

encoding

encoding is only allowed for W fields. *encoding* must be a Unicode encoding. If *encoding* is specified, the field value is converted to the specified encoding before selecting the specified bytes from the field value.

Superdescriptor Standard Length and Format

The description of *format* provides information concerning the standard format of a superdescriptor where all components are unpacked or at least one component is Unicode.

The format is alphanumeric if at least one parent field is alphanumeric, otherwise it is binary.

The format of a superdescriptor can only be specified if all of the parent fields are unpacked, in which case only unpacked and binary can be specified: the default is binary. If not all parent fields are unpacked, the format is alphanumeric if at least one parent field is alphanumeric or Unicode,

otherwise it is binary. If not all parent fields are unpacked, the format is alphanumeric if at least one parent field is alphanumeric or Unicode, otherwise it is binary.

The superdescriptor's standard length is defined by the sum of its elements and is used by Adabas while processing search commands. For example, a search buffer containing only a superdescriptor name, without length override, will use this standard length.

Superdescriptors with Unicode Parent Fields

If encoding has been specified for a superdescriptor parent field, the superdescriptor element is derived from the W field value converted to the specified encoding. If encoding has not been specified for a superdescriptor element derived from a W field, the value of the superdescriptor element is created from the internal encoding of the W field (UTF-8). A conversion to and from the user encoding defined for the user session is done superdescriptor element by superdescriptor element – the conversion is only done for superdescriptor elements for which encoding has been specified. It is not permitted to specify different encodings for the same superdescriptor.

Caution:

1. If you use a superdescriptor with format W, the superdescriptor value is generally not a valid Unicode field, because the superdescriptor can contain elements that are not Unicode fields, and it may happen that a superdescriptor element derived from a Unicode field may begin or end in the middle of a character.
2. If you want to use a UTF-16 or UTF-32 encoding, it is strongly recommended to always specify UTF-16BE or UTF-32BE, but not UTF-16LE or UTF-32LE. The expected search order is only achieved with the big endian encodings, because the sort order for Unicode elements is alphanumeric.
3. If you use superdescriptors with a W field parent that has a user encoding different from the encoding specified for the Unicode superdescriptor elements, you may get incorrect or undefined results. For example, assume you have defined a superdescriptor element FN(1,2,UTF-16BE) to include the first character of the field in the superdescriptor, and the user encoding is UTF-8. If you try to search for a value where the first character of FN is a 3-byte UTF-8 character, the value in the search buffer contains only a part of the character. =>It is not possible to convert the superdescriptor element from UTF-8 to UTF-16.
4. If you read a superdescriptor with a W field parent that has a length > the superdescriptor length, the following rules are used for padding:
   - If the last parent field is a W field, the W field is extended until the end byte according to the specified length.
   - If the last parent field is not a W field, the superdescriptor is padded with A field blanks.
5. It makes a difference whether you specify a superdescriptor parent with encoding UTF-8 or without encoding: only if you explicitly specify encoding UTF-8, will a conversion to or from the user encoding be performed when you use the superdescriptor in an Adabas call.

6. If you explicitly specify the format when you access a superdescriptor, it must be the format of the superdescriptor. However, the processing of the superdescriptor is the same, independent of the format used for the superdescriptor.

Superdescriptors Containing Binary Parent Fields

If a superdescriptor contains binary parent fields (without the HF option), the value of the superdescriptor depends on the platform on which it is used:

- on a high-order first platform, the binary components are defined high-order first.
- on a low-order first platform, the binary components are defined low-order first.

However, the collation of the superdescriptor on low-order first platforms is the same as on high-order first platforms. Although in this respect there is a difference to normal values, the values are handled like other values of the same format. If, for example, you specify a superdescriptor with the A format in the search buffer with a length less than the superdescriptor length, the value is padded with blanks in order to get the complete superdescriptor value.

Examples of Superdescriptor Definitions

The following definitions are used in the next two examples:

```
01,LN,40,W,DE,NU    ;Last-Name
01,FN,40,W,MU,NU    ;First-Name
01,ID,4,B,NU        ;Identification
01,AG,3,U           ;Age
01,AD,PE            ;Address
  02,CI,20,A,NU     ;City
  02,ST,20,A,NU     ;Street
01,FA,PE            ;Relatives
  02,NR,20,A,NU     ;R-Last-Name
  02,FR,20,A,MU,NU  ;R-First-Name
```

Example

```
Superdescriptor definition:   SD = LN(1,4),ID(1,2),AG(2,3)
```

Superdescriptor SD is to be created. The values for the superdescriptor are to be derived from bytes 1 to 4 of field LN (counting from left to right), bytes 1 to 2 of field ID (counting from the low-order byte to the high-order byte), and bytes 2 to 3 of field AG (counting from right to left). Because no encoding has been specified for field LN, the internally-used encoding UTF-8 is kept. All values are shown in hexadecimal. In the following, the internal value shows how the value is represented internally to control the collating sequence of the values, the high-order (h-o) first value shows the representation of the value in the record buffer or value buffer on a high-order first platform, and the low-order (l-o) first value shows the representation of the value in the record buffer or value buffer on a low-order first platform.

```
LN             ID                          AG      SD

464C454D494E47 0x862143 (logical value)    303433  464C454D21433034 (internal)
               00862143 (h-o first)                464C454D21433034 (h-o first)
               43218600 (l-o first)                464C454D43213034 (l-o first)

4D4F52524953   0x2461866 (logical value)   303338  4D4F525218663033 (internal)
               02461866  (h-o first)               4D4F525218663033 (h-o first)
               66184602  (l-o first)               4D4F525266183033 (l-o first)

5041524B4552   00000000                    303336  No value is stored with index

202020202020   0x432144  (logical value)   303030  No value is stored with index
               00432144  (h-o first)
               44214300  (l-o first)

414141414141   0x144     (logical value)   313131  4141414101443131 (internal)
               00000144  (h-o first)               4141414101443131 (h-o first)
               44010000  (l-o first)               4141414144013131 (l-o first)
```

The format for SD is alphanumeric since at least one element (LN) is defined with W format, and no explicit format has been specified.

If you specify a truncated superdescriptor value by specifying the following in the search buffer:

```
SD,5
```

then a value in the search buffer

```
464C45D21
```

is padded with blanks to get the complete superdescriptor value:

```
464C45D21202020
```

If this value has been specified on a high-order first platform, it is also the internal value that is used to resolve the query. If the value has been specified on a low-order first platform, the corresponding internal value is:

```
464C454D20212020
```

Example

```
Superdescriptor definition:   SY,W = LN(1,8,UTF-16BE),FN(1,2,UTF-16BE)
```

Superdescriptor SY is to be created from fields LN and FN (which is a multiple-value field). All values are shown in character format. The format is W.

```
          LN                                  FN                        SY
          FLEMING                             DAVID                     FLEMD
UTF-16BE: 0046 004C 0045 004D 0049 004E 0047  0044 0041 0056 0049 0044  0046 004C ↩
0045 004D 0044
          WILSON                              JOHN                      WILSJ
                                              SONNY                     WILSS
UTF-16BE: 0057 0049 004C 0053 004F 004E004A   004F 0048 004E      0057 0049 004C ↩
0053 004A
                                             0053 004F 004E 004E 0059  0057 0049 ↩
004C 0053 004E
```

As long as all values consist only of 1- or 2-byte UTF-8 characters, you can also work with the user encoding UTF-8. Then the superdescriptor value created for FLEMING, DAVID is converted to 46 4C 45 4D 20 20 20 20 44. Also, if you create a superdescriptor value in an application program from the field values, it works as expected: The value created is 46 4C 45 4D 49 4E 47 20 44 41. The first element of the superdescriptor value is converted to 0046 004C 0045 004D 0049 004E 0047 0020 and then truncated to 0046 004C 0045 004D. The second element of the superdescriptor is converted to 0044 0041 and then truncated to 0044. This means that the converted superdescriptor value is to 0046 004C 0045 004D 0044 - as expected.

However, if there are values containing 3-byte UTF-8 characters, working with user encoding UTF-8 will cause problems!

Example

```
Field Definitions:

01,PN,6,U,NU
01,NA,20,A,DE,NU
01,DP,1,B,FI
Superdescriptor Definition:   SZ = PN(3,6),DP(1,1)
Source Field Values           SZ Values
------------------            ---------
(shown in hex)                (shown in hex)

PN             DP             SZ

303234363732   04             3032343604
383430333938   00             3834303300
303030303131   06             3030303006
303030303031   00             3030303000
```

The format of SZ is binary because no element is derived from a source field defined with A format. A null value is stored for the last value shown because the superdescriptor format is binary and the first value contains unpacked zeros (hexadecimal value '30') and not binary zeros (hexadecimal value '00').

Example

```
Field Definitions:

01,PF,4,P,NU
01,PN,2,P,NU
Superdescriptor Definition:   SP = PF(3,4),PN(1,2)
Source Field Values           SP values
--------------------          ---------
(shown in hex)                (shown in hex)

PF             PN             SP

0002463C       003C           0002003C
0000045C       043C           0000043C
0032464C       000C           No value is stored with index
0038000C       044C           0038044C
```

The format of SP is binary since no element is derived from a source field defined with A format.

Example

```
Field Definitions:

01,AD,PE
02,CI,4,A,NU
02,ST,5,A,NU
Superdescriptor Definition:   XY = CI(1,4),ST(1,5)
Source Field Values           XY values
--------------------          ---------

CI             ST             XY

(1st occ.)     (1st occ.)
BALT           MAIN           BALTMAIN

(2nd occ.)     (2nd occ.)
CHI            SPRUCE         CHI SPRUCE

(3rd occ.)     (3rd occ.)
WASH           11TH           WASH11TH

(4th occ.)     (4th occ.)
DENV           <null value>   No value stored with index
```

The format of XY is alphanumeric since at least one element is derived from a source field which is defined with A format.

Phonetic Descriptor

A phonetic descriptor can be defined in order to perform phonetic searches. The use of a phonetic descriptor in a FIND command returns all of the records with similar phonetic values. The phonetic value for a phonetic descriptor is based on the first 20 bytes of the source field value. Only upper/lower case alphabetic values are allowed; numeric values, special characters and blanks are ignored.

Phonetic descriptors may be defined after the last field definition. Phonetic descriptors may appear before and/or after any subdescriptor or superdescriptor definitions.

Phonetic Descriptor Definition Syntax

```
pn = PHON(fn)
```

pn
The name of the phonetic descriptor. The naming conventions as described previously for Adabas names must be observed.

PHON(fn)
The literal PHON followed by the name of the source field to be phoneticized.

The source field may be an elementary or a multiple-value field and must be defined with alphanumeric format. The source field may or may not be a descriptor. A subdescriptor or superdescriptor may not be specified.

The source field may be contained within a periodic group.

Example

```
Source Field Definition:      01,AA,20,A,DE,NU

Phonetic Definition:          PA = PHON(AA)
```

Hyperdescriptor

A hyperdescriptor is a descriptor whose value is based on a user-supplied algorithm.

The values are based on algorithms coded in special user exits (hyperexit 1 to 255). Each exit may handle multiple hyperdescriptors. Each hyperdescriptor must be assigned to a hyperexit.

The hyperexit is called whenever a hyperdescriptor value is to be generated by the Adabas nucleus, or by the ADAINV, ADACMP or ADAULD utility.

One or more values may be returned depending on the options (PE, MU) assigned to the hyperdescriptor. The original ISN assigned to the input value(s) may be changed.

The format, the length, and the options of a hyperdescriptor are user-defined. They are not taken from the parent fields defined by the hyperdescriptor specification.

A search using a hyperdescriptor value is performed in the same manner as that for standard descriptors.

The user is responsible for creating correct hyperdescriptor values. There is no standard way to check the values of a hyperdescriptor for completeness against the Data Storage. The user must set the rules for each value definition, and check the value for correctness.

If a hyperdescriptor is defined as packed or unpacked format, Adabas will check the returned values for validity.

Please refer to the chapter User Exits and Hyperexits for more information about hyperdescriptors.

Hyperdescriptor Definition Syntax

```
hy-name,length,format[,option... = HYPER(exit_number,parent_field
                                   [,parent  field...])
```

hy-name
The name to be used for the hyperdescriptor. The naming conventions as described previously for Adabas names must be observed.

length
The default length of the hyperdescriptor.

format
The format of the hyperdescriptor. The following formats are supported:

Format	Maximum Length
Alphanumeric (A)	253 bytes
Binary (B)	126 bytes
Fixed Point (F)	4 bytes (always 4 bytes)
Floating Point (G)	8 bytes (always 4 or 8 bytes)
Packed Decimal (P)	15 bytes
Unpacked Decimal (U)	29 bytes

option

The options to be assigned to the hyperdescriptor. The following options may be used together with a hyperdescriptor:

Option	Meaning
HE	Search value generation: allowed only if the number of parent fields = 1. You must specify not the internal search value, but rather the corresponding parent field value. Adabas then calls the hyperexit to convert the value to the internal search value.
MU	Multiple-value descriptor
NU	Null value suppression
PE	Periodic group index usage
UQ	Unique descriptor

exit_number

The hyperexit number to be assigned to the hyperdescriptor. This number will be used by the Adabas nucleus and utilities to determine the hyperdescriptor user exit to be called.

parent field

The names of between one and 20 elementary fields. The field names and addresses are passed to the user exit.

Examples of Hyperdescriptor Definition

The following definitions are used for this example:

```
01,LN,20,A,DE,NU    ;Last-Name
01,FN,20,A,MU,NU    ;First-Name
01,ID,4,B,NU        ;Identification
01,AG,3,U           ;Age
01,AD,PE            ;Address
  02,CI,20,A,NU     ;City
  02,ST,20,A,NU     ;Street
01,FA,PE            ;Relatives
  02,NR,20,A,NU     ;R-Last-Name
  02,FR,20,A,MU,NU  ;R-First-Name
```

Example

```
Hyperdescriptor definition: HN,60,A,MU,NU=HYPER(2,LN,FN,FR)
```

Hyperexit 2 is assigned to this hyperdescriptor, and the name is HN.

The hyperdescriptor length is 60, the format is alphanumeric. The hyperdescriptor is a multiple-value (MU) descriptor with null suppression (NU).

The values for the hyperdescriptor are to be derived from the fields LN, FN and FR.

Example

```
Hyperdescriptor definition: SN,20,A,HE,NU=HYPER(3,LN)
```

Hyperexit 3 is assigned to this hyperdescriptor, and the name is SN.

The hyperdescriptor length is 20, the format is alphanumeric with null suppression (NU). The hyperexit is called to perform a search value generation (HE) for search and read commands that use a search and value buffer.

The value for the hyperdescriptor is to be derived from the field LN.

Collation Descriptor

A collation descriptor is a descriptor that is based on an ICU collating key for a Unicode field, where the ICU collating key is a binary string produced from the original character string by applying a Unicode Collation Algorithm and language-specific rules. When you perform a binary comparison between the collating keys produced this way for character strings, you perform a comparison between the strings that is appropriate to your locale.

Notes:

1. Collation descriptor values are truncated to 1144 bytes. This means that, in some cases, long values can appear to be equal when in fact they are different. Also, collation keys can be much longer than the values from which they are derived.
2. Until Adabas Version 6.4, ICU 3.2 was used. From Adabas Version 6.5 on ICU 5.4 is used by default. ICU 3.2 is supported for existing descriptors in parallel. Please refer to the section*Universal Encoding Support (UES)* for further information about ICU and the versions supported.

Collation Descriptor Definition Syntax

```
col-name [,max_length] [,LA|L4] [,HE] [,UQ] = ↩
COLLATING(parent_field[,collation_attribute]...)
```

col-name
The name to be used for the collation descriptor. The naming conventions as described previously for Adabas names must be observed.

max_length
The maximum number of bytes that are stored as a descriptor value. If the collation key derived from the parent field is larger, the collation key is truncated. The default and maximum value is the maximum descriptor value length (1144).

LA, L4
If you specify one of these options, the length indicator is 2 bytes (LA option) or 4 bytes (L4 option) long if you access the descriptor with variable length.

HE
If you specify this option, you must specify the corresponding parent field value in the value buffer for search operations, rather than the internal collation key. It is not possible to read the descriptor values (L9 command).

If this option is not specified, you can specify either the internal collation key or the corresponding parent field value in the value buffer, depending on the search buffer. In this case, it is possible to read the descriptor values (L9 command).

Notes:

1. In most cases, you won't want to handle internal collation keys, an exception being if you also use ICU in your application programs. Therefore you should usually specify the HE option.
2. If you don't use the HE option, you should remember that collation keys are much larger than their parent fields (4 times the length of the parent value is a typical length for a collation key). This means that one byte is often not sufficient for the length of the collation key, although the parent value is defined without the LA/L4 option, and therefore it is recommended to specify either the LA or the L4 option for the collation descriptor. However, larger collation keys are also stored in the index without the LA or L4 option, but they cannot be read with variable length in an L9 command - trying to do so will result in an Adabas response code 55.

UQ
A collation descriptor can be defined as a unique descriptor.

parent_field
The name of the source field from which the collation descriptor is to be derived. It must have the format W.

collation_attribute

All collation attributes are optional, and they can be specified in any order. The following collation attributes can be specified:

Locale string

One of the locales supported by ICU. This usually is a 2 character ISO-3166 language code. It can be followed by "@" and "collation=" <collation specifier>. This string must be enclosed in single quotes. Example: 'de@collation=phonebook'

The default is '' (empty string). Collating keys are then compatible with the Unicode Default Collation Table (this is language-independent, but provides good results for many languages).

Collation strength

You can specify one of the following keywords: PRIMARY, SECONDARY, TERTIARY, QUARTERNARY, IDENTICAL. The value specified represents the comparison levels. See references 1 and 2 below for further information.

If you specify PRIMARY, case and diacritic differences are ignored. SECONDARY means that case differences are ignored, and punctuation is ignored if you specify TERTIARY. QUARTERNARY allows you to distinguish between words with and without punctuation, e.g. with TERTIARY "ab" = "a-b" and with QUARTERNARY "ab" < "a-b". If you specify IDENTICAL, only words with the same canonical decomposition are considered as equal.

The default is TERTIARY.

case-first option

You can specify one of the following keywords: UPPERFIRST or LOWERFIRST.

If you specify UPPERFIRST, uppercase letters will be sorted before lowercase letters, e.g. 'AB' > 'ab'.

If you specify LOWERFIRST, lowercase letters will be sorted before uppercase letters, e.g. 'ab' > 'AB'.

If not specified, the case-first processing is undefined.

alternate_option

You can specify one of the following keywords: SHIFTED or NON_IGNORABLE.

These keywords affect the sorting sequence for punctuation characters such as space or hyphen: for example, the words "bi-weekly" and biweekly" will be sorted close together if you specify SHIFTED, and they will not be sorted close together if you specify NON_IGNORABLE. See references 1 and 2 below for further information.

The default is NON_IGNORABLE.

case_level_option

You can specify one of the following keywords: CASELEVEL or NO_CASELEVEL.

If you specify CASELEVEL, an additional case level is formed between secondary and tertiary. Currently, the case level is used for Japanese, but it could also be used in other situations, such as Pinyin. See reference 2 below for further information.

The default is NO_CASELEVEL.

french_option
You can specify one of the following keywords: FRENCH or NO_FRENCH.

The setting of this option determines whether or not diacritics will be sorted as in French.

The default is NO_FRENCH.

normalization_option
You can specify one of the following keywords: NORMALIZATION or NO_NORMALIZATION.

The setting of this option determines whether or not Unicode canonical equivalence is to be taken into account. Even if NO_NORMALIZATION is set, ICU will still produce correct results for non-normalized text for most world languages. However, languages that can use two or more diacritic marks in one character (e.g. Hebrew, Thai or Vietnamese) require this option to be set if the input is not normalized according to Unicode normalization form D. See reference 2 below for further information.

The default is NO_NORMALIZATION.

Example

```
Collation descriptor definition: C1,HE,UQ=COLLATING(W1,'en',PRIMARY) ↩
```

A unique collation descriptor is defined with HE option, language is English, and the collation strength is PRIMARY.

```
Collation descriptor definition: C2,HE=COLLATING(W2,'de@collation=phonebook') ↩
```

A collation descriptor is defined with HE option, language is German, and the phonebook order is to be used. The collation strength is default (TERTIARY).

References on ICU Collations

1. Mark Davis, Ken Whistler: "Unicode Technical Standard #10, Unicode Collation Algorithm" (**http://www.unicode.org/reports/tr10/**)
2. International Components for Unicode homepage (**https://www-01.ibm.com/software/globalization/icu/**)

Referential Constraints

A referential constraint ensures referential integrity between two keys. Keys can be descriptors, superdescriptors or ISNs. Referential integrity means that for every value in a descriptor called “foreign key”, there must be a value in a descriptor called “primary key" in a primary file. The primary key must be defined as unique and the options NC and NN must be set. For the foreign key, the option NC must be set. A pair of primary and foreign key must have the same format. ISNs can be used as primary keys and the corresponding foreign key must be binary. If primary and foreign keys are superdescriptors, then

- The corresponding key must be a superdescriptor;
- The superdescriptors must have the same number of parent fields;
- The corresponding parent fields must have the same format;
- No parent field may occur twice in a superdescriptor;
- No parent field may occur in more the one foreign key;
- All parent fields must have the option NC;
- All parent fields of the primary key must have the option NN;
- The primary key superdescriptor must be unique.

You can specify a referential action that is executed on the foreign key record if a primary key value is modified.

The referential constraint is added to the file to which the foreign key belongs.

Referential Constraint Syntax

```
Constraint-name = REFINT(foreign-key, primary-file,
     primary-key[/referential-action[,referential-action]])
```

foreign-key
The name of the foreign key field

primary-file
The number of the file to which the primary key belongs.

primary-key
The name of the primary key.

referential-action
One of the following keywords:

Keyword	Description
DC	On delete cascade: If a record in the primary file is deleted, the records containing the primary key as a foreign key are also deleted. If these records also contain a primary key of a referential constraint, then the corresponding referential action is also performed for these keys.
DX	On delete no action (default): If a record in the primary file is deleted, no further records that contain the primary key as a foreign key may still exist. Otherwise the delete operation fails with an Adabas response code 196.
DN	On delete set NULL: If a record in the primary file is deleted, the foreign key field is set to NULL in all records that contain the primary key as a foreign key. This option is not allowed if the foreign key field is defined with the option NN.
UC	On update cascade: If a primary key is updated in a record in the primary file, the foreign key is also set to the new value of the primary key in the records that contain the primary key as a foreign key. If the foreign key is also the primary key of a referential constraint, then the corresponding referential action is also performed for these keys.
UX	On update no action (default): If the primary key in a record in the primary file is updated, no further records that contain the old primary key value as a foreign key may still exist. Otherwise the update operation fails with an Adabas response code 196.
UN	On update set NULL: If the primary key in a record in the primary file is updated, the foreign key field is set to NULL in all records that contain the old primary key value as a foreign key. This option is not allowed if the foreign key field is defined with the option NN.

You can specify up to one delete action and up to one update option for a referential constraint. For constraints that refer to ISNs as primary key, only the actions delete cascade and update no action (DC,UX) are possible.

Example:

```
Primary key definition in file 9:      1, AA,8,A,DE,UQ,NC,NN
Foreign key definition in file 12:     1, AC,8,A,DE,NC

HT=REFINT(AC,9,AA)
HT=REFINT(AC,9,AA/UC)
HT=REFINT(AC,9,AA/UC,DN)
```

6 Defining Descriptors

ADAINV Processing Considerations

Establishing a New Descriptor

Defining a new descriptor leads to modifications in the Field Definition Table (FDT) and results in the creation of an inverted list. New Main Index and Normal Index blocks are required to store the inverted list entries for the new descriptor. ADAINV allows any number of descriptors to be established within the same run.

ADAINV builds the new Normal Index and Main Index on a descriptor by descriptor basis. During this pass, the linking entries are still missing in the Upper Index and none of the new inverted lists can, therefore, be accessed. When the Normal Index and Main Index of all the new descriptors have been built, the FDT is updated and corresponding entries are added to the Upper Index.

Loading of Normal Index and Main Index

All of the data records in the file have to be read in order to build the new Normal Index and Main Index. Within each record, the field associated with the new descriptor is used to generate a descriptor value and it's ISN. These values are sorted according to ascending descriptor values and ISNs. The output of the sort is used to build the new Normal Index and Main Index. Descriptors defined with the unique option are checked to ensure that the Normal Index contains only one ISN per descriptor value. If more than one ISN is found, the conflicting ISNs are written to the error log, the unique flag is reset within the FDT and processing continues if UQ_CONFLICT is set to RESET. If UQ_CONFLICT is set to ABORT or is omitted, ADAINV aborts.

Besides sorting the descriptor values, reading the data records is very time-consuming because of the numerous I/Os. Therefore, if a large number of descriptors are to be established in one run, ADAINV tries to minimize the number of passes required to read through the data storage. In the first pass through the data storage, the values for one descriptor are directly passed to the sort. The values of two additional descriptors, if they exist, are written to the TEMP data set, and all other values of the remaining descriptors, their total sizes and quantities are accumulated. This accumulation of data optimizes the remaining passes through the data storage. The greater the number of descriptors using the TEMP in parallel during each pass, the faster the inversion will be. ADAINV displays the total number of passes required at the end of the run.

All index blocks are filled in accordance with the padding factor specified when the file was loaded. New index blocks are taken from the existing extents as required. When these blocks are exhausted, an automatic extension is carried out in accordance with the rules described for the mass update utility ADAMUP.

Processing continues as described above if the extension is successful, otherwise ADAINV terminates with an error message.

Updating the Upper Index

Whereas the Normal Index and Main Index are organized on a descriptor by descriptor basis, the Upper Index, index level 3 and higher, contains all descriptors. In order to link in the new Main Index, an entry must be made in the Upper Index for each new Main Index block. The new entries can be added in two ways:

- If the updates only affect level 3 blocks, the new entries are inserted directly into the existing Upper Index.
- If block splitting or updates in levels higher than 3 become necessary, the whole Upper Index is rebuilt. The padding factor specified when loading the file is reestablished. All old index blocks and pre-allocated blocks are used before additional blocks are allocated. If additional blocks are required, the procedure described for Normal Index and Main Index loading is used.

Releasing a Descriptor

Releasing a descriptor leads to modifications in the Field Definition Table (FDT) and results in the elimination of an access path. All Main Index and Normal Index blocks that contain the inverted list entries for the descriptor are released, but cannot be reused. Any number of descriptors may be released in one run.

Updating the Upper Index

When a descriptor is released, corresponding entries have to be removed from the Upper Index. These entries can be removed in two ways:

- If the updates only affect level 3 blocks, the new entries are deleted directly from the existing Upper Index.
- If blocks become empty or updates in levels higher than 3 are necessary, the whole Upper Index is rebuilt. The padding factor specified when loading the file is re-established. All old index blocks and pre-allocated blocks are used before additional blocks are allocated. If additional blocks are required, the procedure described for Normal Index and Main Index loading is used.

Releasing Main Index and Normal Index

In principle, a descriptor can be released just by removing the corresponding entries from FDT and Upper Index. Disabling the link to the index level below virtually deletes the Main Index and Normal Index. Although not mandatory, ADAINV physically clears all Normal Index and Main Index blocks of the old descriptor. The resultant overhead is compensated for by data security and improved performance of the backup utility ADABCK, since empty blocks are not dumped.

Checking the Integrity of Inverted Lists

Inverted lists are maintained by Adabas for each elementary, sub-, super-, hyper- and phonetic descriptor defined within a file. In order to guarantee their integrity, it must be ensured that

- each ISN in the inverted list is associated with an existing data record and that this data record is the correct one;
- each record in the data storage is represented in the inverted list by its ISN and the descriptor value entries generated.

When verifying a descriptor, ADAINV simulates loading of the normal index and matches the output from the sort against the content of the inverted list. This checks both of the points mentioned above in one run and detects uniqueness conflicts. All inconsistencies found will be reported. The file remains unchanged.

Rejected Data Records

Any records rejected by ADAINV are written to the ADAINV error file. The contents of this error file should be displayed using the ADAERR utility. Do not print the error file using the standard operating system print utilities, since the records contain unprintable characters.

Please refer to the ADAERR utility in the Adabas Utilities Manual for further information.

7 Using Utilities

Where appropriate, specific sections are used when there are substantial handling differences between platforms (UNIX and Windows).

Assigning Input and Output Devices

All commands to utilities are read from *stdin*or *SYS$INPUT*, and the output is directed to *stdout* or *SYS$OUTPUT*. The standard input and output may be directed to files that are normally supported by the operating system. By default, the standard output does not include the utility parameters specified, but if you set the environment variable ADAPARLOG to YES, the parameter specifications are copied to stdout/SYS$OUTPUT, provided the parameters are not specified interactively.

If the utility produces one or more output files, the environment variable/logical name corresponding to each output file must be set to a legal file name, before the utility is started. For detailed information concerning the assignment of environment variables/logical names, refer to the utilities in question. The output files produced by each utility and the environment variable/logical name assignments for these files are described near the beginning of each utility chapter. If the environment variable/logical name for an output file is not set before the utility is started, the output file is created in the current directory, and the name of the output file is the same as the environment variable/logical name. If the environment variable/logical name for a command log file or a protection log file is not set before the utility is started, the output file is created in the current directory, and the name of the output file is the same as the environment variable/logical name with a sequence number appended.

Several utilities also require environment settings for user exits and hyperexits. For more details, refer to the section User Exits and Hyperexits.

Executing a Utility (UNIX)

The utilities may be executed interactively or in the background.

Prerequisites

You must have the necessary permissions in order to execute a utility. The installation sets the permissions as follows: utilities may be executed by the Software AG products administrator user, e.g. sag, and by all users that belong to the corresponding group, e.g. sag. If this group is not your current group, it is recommended that you change the group with the `newgrp` command before you execute a utility (including ADANUC); failing to do so may lead to other users encountering permission problems because the group for IPC resources created by the user is the current group of the user who executes the utility, and some IPC resources allow access for only the group.

Before you execute a utility, you must set the required environment variables if the names of the files to be used differ from the default values (see the section “Environment Variables” in this chapter).

The program dbgen generates a file called assign.bsh (for Bourne shell and Korn shell) or assign.csh (for C shell) when they create a new database. This file can be sourced in the shell to set the environment variables for the container files, command log and protection log. To source this file, change your working directory to your database directory and enter:

```
.  ./assign.bsh
```

If you do not set the environment variables for container files externally, the utilities extract the settings themselves via the configuration files ADABAS.INI and DBxxx.INI, as described in the Adabas Extended Operation section.

Utility input lines contain control statements that consist of strings defining the settings of parameters, a function to be executed or both.

All functions and parameters are described in the documentation, as well as in help messages.

The utilities are located in the directory $ADAPROGDIR. If this directory is defined in the PATH environment variable, you can execute the utility directly by specifying its name at the operating system prompt, otherwise you have to precede the utility name by “$ADAPROGDIR”. The following examples assume that the directory name $ADAPROGDIR is defined in the PATH environment variable.

Executing a Utility Interactively

You execute a utility interactively by specifying its name followed by carriage return at the operating system prompt.

When a utility starts, it displays an informational message consisting of date, time and version number. It then prompts with its name and a colon for input of control statements:

```
%ADAREP-I-STARTED, 18-JUL-2005 11:39:49, Version 5.1.1
adarep:
```

In interactive mode, the input line starts directly after the utility prompt.

You can extend the input over several physical input lines by terminating each line with a backslash (“\”) followed by carriage return. The utility outputs the prompt “>” at the start of each continuation line.

Executing a Utility at Call Level

The control statements also be specified at the call level, e.g. for the utility adarep:

```
adarep dbid=20 summary
```

The control statements can also be redirected from an input file. If, for example, the file rep.in contains the lines:

```
dbid=20
summary
```

then you can run adarep with these two control statements by specifying:

```
adarep <rep.in
```

The output produced by the utility usually goes to *stdout*, which means that it can be redirected. So, for example, to redirect the output of the sample adarep call to the file rep.out, specify:

```
adarep <rep.in >rep.out
```

Notes:

1. Please be aware that the UNIX shells have a special handling for some characters such as parentheses, quotes and double quotes. This means that you must change the specification of the parameters accordingly if the parameter values contain one or more of these characters.
2. Blanks are equivalent to a line feed in interactive input - if a blank should appear in a parameter value, it must be preceded by a backslash or occur between quotes or double quotes.
3. Setting the environment variable ADAPARLOG to YES may help you to find errors in the parameter specification - it displays the parameters as specified interactively.

Examples

```
adadcu fields 'NEW RECORD',AA,AB end_of_fields
```

Invalid, because it is equivalent to

```
adadcu
fields
NEW RECORD,AA,AB
end_of_fields
```

The quotes are missing.

```
adadcu fields \'NEW RECORD\',AA,AB end_of_fields
```

Invalid, because it is equivalent to

```
adadcu
fields
```

```
'NEW
RECORD',AA,AB
end_of_fields
```

The literal must be specified on one line.

```
adadcu fields \'NEW\ RECORD\',AA,AB end_of_fields
```

Valid, this is equivalent to

```
adadcu
fields
'NEW RECORD',AA,AB
end_of_fields
```

```
adadcu "fields 'NEW RECORD',AA,AB end_of_fields"
```

Invalid, because it is equivalent to

```
adadcu
fields 'NEW RECORD',AA,AB end_of_fields
```

fields and end_of_fields must be specified on separate lines.

```
adadcu fields "'NEW RECORD'",AA,AB end_of_fields
```

or

```
adadcu fields "'NEW RECORD',AA,AB" end_of_fields
```

Valid, both are equivalent to

```
adadcu
fields
'NEW RECORD',AA,AB
end_of_fields
```

Switching Parameter Input from Command Line to Standard Input

It is possible to switch the parameter input for a utility from the command line to standard input by entering a '+' character (plus sign) after the last parameter specified in the command line.

Example:

```
adafdu dbid=35 file=36 + <employee.fdu
```

Executing a Utility (Windows)

The utilities may be executed interactively or in the background.

In order to be able to execute Adabas utilities of an installed Adabas version, some environment variables must be set. This is done when you select **Adabas Server Command Prompt**.

If you want to execute Adabas commands from the standard command prompt, you must configure a 'local' Adabas Server Environment as described in *Completing the Installation*.

- You can set the environment variables required for Adabas as system environment variables by selecting **Set Adabas System Environment**. You can delete these system environment variables again by selecting **Unset Adabas System Environment**.
- If you don't want to set the environment variables as system environment variables, you can execute the command *startenv.cmd* in the bin folder of the current Adabas version. You can find out the name of the folder by right-clicking on the **Adabas Command Prompt** and selecting **Properties**.

Before you execute a utility offline, you must set the required environment variables if the names of the files to be used differ from the default values (see the section "Environment Variables" in this chapter).

If you do not set the environment variables for container files externally, the utilities extract the settings themselves via the configuration files ADABAS.INI and DBxxx.INI, as described in the Adabas Extended Operation documentation.

Utility input lines contain control statements that consist of strings defining the settings of parameters, a function to be executed or both.

All functions and parameters are described in this manual, as well as in help messages.

The utilities are located in the subdirectory "Adabas" of the installation directory. If this directory is defined in the PATH environment variable, you can execute the utility directly by specifying its name at the operating system prompt; otherwise you have to precede the utility name by "%ADAPROGDIR%". The following examples assume that the directory name %ADAPROGDIR% is defined in the PATH environment variable.

Executing a Utility Interactively

You execute a utility interactively by specifying its name followed by carriage return at the operating system prompt.

When a utility starts, it displays an informational message consisting of date, time and version number. It then prompts with its name and a colon for input of control statements:

```
%ADAREP-I-STARTED, 18-JUL-2005 11:39:49, Version 5.1.1
adarep:
```

In interactive mode, the input line starts directly after the utility prompt.

You can extend the input over several physical input lines by terminating each line with a backslash ("\") followed by carriage return. The utility outputs the prompt ">" at the start of each continuation line.

Executing a Utility at Call Level

The control statements also be specified at the call level, e.g. for the utility adarep:

```
adarep dbid=20 summary
```

The control statements can also be redirected from an input file. If, for example, the file rep.in contains the lines:

```
dbid=20
summary
```

then you can run adarep with these two control statements by specifying:

```
adarep <rep.in
```

The output produced by the utility usually goes to *stdout*, which means that it can be redirected. So, for example, to redirect the output of the sample adarep call to the file rep.out, specify:

```
adarep <rep.in >rep.out
```

Note: Blanks are equivalent to a line feed in interactive input - if a blank should appear in a parameter value, it must occur between double quotes.

Examples

```
adadcu fields 'NEW RECORD',AA,AB end_of_fields
```

Invalid, because it is equivalent to

```
adadcu
fields
'NEW
```

```
RECORD',AA,AB
end_of_fields
```

The literal must be specified in one line.

```
adadcu "fields 'NEW RECORD',AA,AB end_of_fields"
```

Invalid, because it is equivalent to

```
adadcu
fields 'NEW RECORD',AA,AB end_of_fields
```

fields and end_of_fields must be specified in separate lines.

```
adadcu fields "'NEW RECORD'",AA,AB end_of_fields
```

or

```
adadcu fields "'NEW RECORD',AA,AB" end_of_fields
```

valid, both are equivalent to

```
adadcu
fields
'NEW RECORD',AA,AB
end_of_fields
```

Switching Parameter Input from Command Line to Standard Input

It is possible to switch the parameter input for a utility from the command line to standard input by entering a '+' character (plus sign) after the last parameter specified in the command line.

Example:

```
adafdu dbid=35 file=36 + <employee.fdu
```

Executing a Utility Remotely

If the option LOCAL_UTILITIES is not set for an active nucleus of a given Adabas database, the database can be accessed remotely by certain Adabas utilities, using SOFTWARE AG's product ENTIRE NET-WORK or NET-WORK ACCESS, provided that the architecture of the local and remote machines is identical (for example, that both machines use byte-swapping). The utilities that can access a database remotely are:

- ADACMP
- ADADBM
- ADAFDU

- ADAREC
- ADAREP
- ADATST
- ADAULD

For ADADBM and ADAULD, only the functions which are allowed when the nucleus is running can be executed remotely.

Note: ADAFDU cannot be executed remotely if you define an ADAM file or if you specify the parameter FORMAT.

Utility Syntax

Functions and Arguments

The arguments of a function are called parameters. Parameters may have various values. A parameter that has only two possible values, which are logically complementary, is called a switch.

Functions are represented by reserved keywords. Three types of functions are used:

- Functions requiring no arguments;
- Functions requiring one argument;
- Functions requiring more than one argument.

A function that does not require an argument is a switch, and can be enabled or disabled by keywords such as LOCK and UNLOCK.

Arguments are of the following type:

- **Strings**
  A string is a sequence of valid ASCII characters with the exception of comma `,', carriage return <CR>, semicolon `;' opening `(' and closing brackets `)', which are all regarded as string delimiters. Embedded blanks and tabs are removed. The backslash `\' can be used as an escape character.

  Most strings are converted to upper case. Instead of the equals sign, it is also possible to specify a colon ':'. The difference is that strings following a colon are not converted to upper case. Although a colon may also be specified for other parameters where the utility syntax description does not allow a colon, it is recommended to use a colon only in these cases; otherwise the results of the parameter specification may be undefined.

Example:

```
adatst: rb=test
adatst: rb
  <<<<<< RECORD BUFFER >>>>>>

00000000    54455354 ........ ........ ........    TEST............
adatst: rb:test
adatst: rb
  <<<<<< RECORD BUFFER >>>>>>

00000000    74657374 ........ ........ ........    test............
```

If you specify “rb=test”, the string “test” is converted to upper case, while it is not converted if you specify “rb:test”.

- **Keywords**
  A keyword is a predefined string.
- **Numbers**
  A number is a string of digits representing an unsigned integer. Numbers can be entered in hexadecimal format by preceding the digits with the two characters “0x”.
- **Number ranges**
  A number range is a number followed by a hyphen (-) and another number.
- **Lists of keywords, strings, numbers or number ranges**
  A list is a sequence of keywords, strings or numbers, separated by commas and enclosed in brackets.

Arguments are specified according to the following rules:

- The first argument is specified after the equal sign `=' following the function keyword;
- Subsequent arguments are specified by using keywords followed by an equal sign and a value.

Examples:

```
adadbm: recover
```

```
adadbm: renumber = (3,14)
```

```
adadbm: reuse = (isn,ds), file = 6
```

```
adarec: regenerate = *
adarec: plog = 654
adarec: checkpoint = synp
adarec: block = 0x1A
```

The list of arguments may be continued in the following lines. In the case of a multi-function utility, the function is executed if this list is complete and correct.

Symbols used in Syntax Diagrams

If an argument is mandatory or has a default value, this is indicated by the characters `M' or `D' in the list of arguments at the beginning of the utility description.

Checking Current Parameter Settings

The current values of all parameters that were set or preset can be displayed by entering an asterisk `*':

```
adarep:*<cr>
```

```
%ADAREP-I-PARSET setting of COUNT=<set>
%ADAREP-I-PARSET setting of DBID=5
%ADAREP-I-PARSET setting of FDT=<set>
%ADAREP-I-PARSET setting of FILES=(10)
```

If a switch is set, it is displayed as "<set>".

Absolute Time

Some utilities (ADACLP, ADADBM, ADAREP) require arguments in the form of an absolute date or date and time string. The string used must correspond to the following absolute format:

```
dd-mmm-yyyy[:hh:mm:ss]
```

For more detailed information about how to use absolute dates and times, please refer to the individual utilities concerned.

Single- and Multi-function Utilities

Depending on the number of functions that can be performed during one run, a utility is called a single-function or multi-function utility.

In a single-function utility, the correctness of each keyword value is checked after it is entered. The completeness of the set of arguments is checked when the utility is executed by pressing CTRL/D (UNIX) on a line by itself or by pressing CTRL/Z followed by a carriage return (Windows). A single-function utility is terminated after the execution of a function.

A multi-function utility accepts sequences of function specifications and/or parameter settings. The sequence must correspond to a certain logical order. A function is executed when this sequence is complete and correct. The execution of a function in a multi-function utility is terminated with a message indicating successful or non-successful completion. The utility itself does not terminate after the execution of a function.

Terminating a Utility

The *QUIT* control statement or its abbreviation "Q" is used to terminate a utility. Also *EXIT* can be used as a synonym for *QUIT*.

Note: Utility control parameters are always converted to upper case. It is, therefore, also possible to specify *QUIT* and *EXIT* in lower case. *QUIT* and *EXIT* can also be specified in the FDT parsing mode, for example, after specifying the FIELDS parameter. However, no conversion to upper case is done in the FDT parsing mode after specifying the LOWER_CASE_FIELD_NAMES parameter; in this case, you must specify *QUIT* or *EXIT* in upper case.

In a single-function utility, *QUIT* aborts the function and terminates the utility. An EOF (CTRL/D on a line by itself [UNIX] or CTRL/Z followed by a carriage return [Windows]) first executes the function and then terminates the utility.

In a multi-function utility, *QUIT* terminates the utility. If entered while setting the function parameters, *QUIT* aborts the function and terminates the utility. However, an EOF tries to execute the function before termination. If some of the parameters required have not been specified, the utility requests their entry before the function is executed.

In all cases of termination, a statistical summary of the IOs made on the input and output files is given and an informational message is issued with utility name, date, time and elapsed time e.g.:

```
%ADAFRM-I-IOCNT,      500  IOs on dataset WORK
%ADAFRM-I-IOCNT,      800  IOs on dataset DATA
%ADAFRM-I-IOCNT,      600  IOs on dataset ASSO
%ADAFRM-I-TERMINATED,   28-JUL-2005 11:39:50, elapsed time: 00:00:51
```

If a utility terminates successfully, the final TERMINATED message is displayed and the program exit status is set to 0, otherwise an ABORTED message is issued and the program exit status is set to a non-zero value. Multi-function utilities always terminate successfully if they are used interactively without input redirection.

In some cases, utilities display the TERMINATED message and deliver a non-zero exit status to indicate an exception, for example the utility ADAVFY if an error has been found, or the utility ADACMP if records were rejected.

Error Handling

Incorrect sections of utility commands are indicated by up arrows in the following line and an error message in a subsequent line. Errors are processed from left to right.

If an error occurs in a sequence of parameter settings, all settings before the error are executed.

```
adadbm: refresh = 30
                  ^
%ADADBM-E-NOTLOAD,  file not loaded
```

The position of the error in the input line is indicated by an up arrow in the following line.

Adabas Sequential Files

Overview

The Adabas nucleus or utilities usually work with sequential files for input and output. In addition to these sequential files, there are also internal files with a uniform internal Adabas format (Protection Log, for example), and external files with a user-defined format (decompressed data, for example).

Internal Adabas sequential files may be directed to the file system, a raw section or a tape device. Some Adabas sequential files can be directed to a named pipe. The following table lists all of the internal Adabas sequential file types, the ADADEV keywords for raw device access, and the corresponding environment variables that can be set to a disk section (UNIX only), tape device or file system. If the environment variable is not defined, Adabas uses a file in the current working directory with the name of the environment variable.

Sequential File Type	ADADEV Keyword	Environment Variable
Protection Log	PLG	DEVPLGx, NUCPLGx(*), PLPLEX, PLPPLGx, RECPLGx
Command Log	CLG	CLPCLGx, DEVCLGx, NUCCLGx
Backup File	BCK	BCK00nx, DEV00nx, ULD00nx
Backup File Copy	BCKOUT	BCKOUTx, DEVOUTx
Compressed Data	DTA	CMPDTAx, DCUDTAx, DEVDTAx, MUPDTAx, ULDDTAx
Descriptor Value Table	DVT	CMPDVTx, DEVDVTx, MUPDVTx, ULDDVTx
ADAMUP Temporary Data	MUPTMP	MUPTMPx
ADAMUP Log (DTA/DVT)	MUPLOG	DEVLOGx, MUPLOGx
Reorder File	ORDEXP	DEVEXPx, ORDEXPx

Sequential File Type	ADADEV Keyword	Environment Variable
Utility Error File	ERR	ERRINx, CMPERRx, DCUERRx, DEVERRx, INVERRx, MUPERRx, RECERRx,

where: n = 1 - 9; x is omitted or is greater than 1.
(*) NUCPLG cannot be on tape.

The following Adabas sequential files must be located in a file system:

Sequential File Type	Environment Variable
ADAMUP ISN List	MUPISN
ADAMUP temporary working space for LOB processing	MUPLOB, MUPLBI
Field Definition Data	FDUFDT, CMPFDT, DCUFDT
Decompressed Raw Data	CMPIN, DCUOUT

The disk-space management utility ADADEV can be used to manage files in raw disk sections (see *ADADEV* for further information [UNIX only]).

Platform Dependencies

Adabas sequential files have different internal formats on high-oder-first and low-order-first platforms, but they are compatible between all platforms with the same byte order.

Examples

Platform 1	Byte Oder	Platform 2	Byte Order	Compatible
Solaris	High-order-first	AIX	High-order first	Yes
LINUX (Intel)	Low-order-first	Windows	Low-order-first	Yes
Solaris	High-order-first	Windows	Low-order-first	No

Only the following files are compatible between all platforms:

- Decompressed data files
- Pure text files, for example FDT files

If you want to exchange date between high-order-first and low-order-first platforms, you should proceed as follows:

1. Unload with ADAULD
2. Decompress with ADADCU
3. Compress with ADACMP
4. Load with ADAMUP

If this is not possible because the decompressed records are too large, you must write a program that reads data from the source database. The program can either write data to an intermediate file in the file system, from where it is read by a second program which writes data to the target database, or it can store the data directly in the target database via Entire Net-Work.

Using Named Pipes

Named pipes can be used to save storage space when the output of a utility is to be processed by another utility or application, or if the input of a utility is output by another utility or application. Named pipes can be used for all Adabas sequential files except MUPDTA, MUPDVT, MUPISN, MUPLBI, MUPLOB, MUPLOG, MUPTMP, NUCPLG, ORDEXP, PLPLEX, PLPPLG, ULD00n.

Named pipes under Windows are transient objects. If an environment variable contains the name of a named pipe (for example \\.\pipe\anyname), the utility that attempts to write to the named pipe will create it implicitly. Then this utility waits for a predefined amount of time until the reading utility comes up and begins to read from the named pipe. If the time elapses without the reading utility coming up, the writing utility aborts with an I/O error.

Multiple Extents For Adabas Sequential Files

If the space allocated to a sequential file becomes exhausted while it is being created by a utility, Adabas attempts to continue the file in another free slot. This applies to all sequential files that are located on raw device if the allocated or pre-allocated space becomes exhausted or if the file system where the sequential file is located becomes full. Multiple environment variables may be used with this technique, with a counter being appended to the name of the environment variable.

Example (C shell)

```
setenv  NUCPLG  /dev/rdsk/c4d0s2
setenv  NUCPLG2  /usr/adabas/PLOG_2
setenv  NUCPLG3  /dev/rdsk/c7d0s2
```

These three devices are used for the Protection Log during a nucleus session.

An alternative method is to set all of the file/device names within one environment variable:

Note: For Unix platforms the delimiter is a blank, and on Windows platforms it is the semicolon character ";".

Example (C shell)

```
setenv NUCPLG "/dev/rdsk/c4d0s2  /usr/adabas/PLOG_2  /dev/rdsk/c7d0s2"
```

Adabas starts with the first environment variable (in this case NUCPLG), or with its default if the environment variable is not set (the default is a file in the current directory with the same name as the environment variable). When the end of the file is reached, or when the file system becomes full (in cases where a file system is used), the file will be closed, and the next environment variable

(NUCLPG2) will be translated and the associated file or disk section opened. Then it allocates free space from that disk section to be used for the next extent. If a subsequent environment variable does not exist (e.g. NUCPLG4), Adabas will wrap around to the first disk section (NUCPLG).

If only one environment variable is defined and assigned to a disk section, the subsequent file extents will all be created in the same disk section. If only one environment variable is defined and assigned to a file in a file system, Adabas cannot continue if the file system becomes full.

When a switch occurs from one file extent to another, the closed extent usually gets renamed by appending the extent number to its name. For example, a file extent called PLG.3 in a disk section will be renamed to PLG.3(1), and a file extent called /tmp/BCK_DB in a file system will be renamed to /tmp/BCK_DB(1). The name of the new file extent contains the new file extent number. Thus, in the example just mentioned, the new file extent in the disk section is named PLG.3(2) and the new file extent in the file system is named /tmp/BCK_DB(2).

Note: The size of the value of an environment variable is limited to 512 bytes or more, depending on the operating system. If you specify a large number of extents in one environment variable, it may happen that the value of the environment variable is truncated, and this will result in an error. In this case, you must specify the extents in separate environment variables.

Calling Utilities

There is one important restriction that applies to using the utilities with multiple extent files: input redirection is not allowed if more than one file extent is required and only one environment variable is set to only one value. In this case, the utility aborts with a message of the following form:

```
%ADABCK-E-SFNOTPO, subsequent file extents not possible with input redirection
```

The following modes will work successfully:

Interactive (UNIX)

```
adabck <RETURN>
dbid=5, dump=4 <RETURN>
<CTRL/D>
```

Interactive (Windows)

```
adabck <RETURN>
dbid=5, dump=4 <RETURN>
<CTRL/Z><RETURN>
```

Interactive, command line

```
adabck dbid=5 dump=4
```

Batch job

```
adabck dbid=5 dump=4 &
```

The utility process will be stopped if a subsequent tape is required. In this case, the user can bring the job into the foreground in order to enter a valid device name and continue processing.

Reading Multiple-Extent Files

When reading sequential files that are split into multiple extents, Adabas checks whether an extent is finished and prompts for the file/device path name of the next extent at which reading is to continue, if just the normal environment variable is given.

Example:

ADAREC regenerates a Protection Log that is spread over 2 sections. The environment variable RECPLG must point to the first extent of protection log 5 (PLG.5(1)). Once it has been processed, ADAREC prompts for the file/device path name of the next extent:

```
 Enter next file/device path name to continue PLG.5(2)
```

The file extent can be either in a disk section or a file system or on a tape device. After opening the subsequent extent, Adabas checks to see whether the given extent is the one it expects. If it is not the correct one, an error message is displayed and the prompt for the correct file/device path name is displayed again. If such a subsequent prompt appears, you can abort further processing by entering “quit”. Situations can arise where no subsequent protection log extent exists although Adabas expects one. In this case, you can enter “close” to terminate the processing of the current protection log file without error.

It is also possible to process the files in one go, without prompting, when using `counted' environment variables. In this case, the same technique is used as when creating the files. The file(s) may be located on raw device, file system or tape device.

Example

C Shell:

```
setenv RECPLG /dev/rmt/0m
setenv RECPLG2 /dev/rdsk/c10d0s2
setenv RECPLG3 /dev/rdsk/c7d0s2
```

Bourne Shell, Korn Shell:

```
RECPLG = /dev/rmt/0m
export RECPLG
RECPLG2 = /dev/rdsk/c10d0s2
export RECPLG2
RECPLG3 = /dev/rdsk/c7d0s2
export RECPLG3
```

Windows:

```
set RECPLG=\\.\TAPE0
set RECPLG2=C:\BCK_2
set RECPLG3=D:\BCK_3
```

Example

An alternative method is to set all of the file/device names within one environment variable.

If the mass update utility ADAMUP uses a data file (DTA) that is split into 2 extents, then the following can be set:

C Shell:

```
setenv  MUPDTA  "/dev/rdsk/c5d0s2  /usr/adabas/DTA_2"
```

Bourne Shell, Korn Shell:

```
MUPDTA  = "/dev/rdsk/c5d0s2  /usr/adabas/DTA_2"export MUPDTA
```

Windows:

```
set MUPDTA=\\.\TAPE0;C:\adabas\DTA_2
```

Tape Device Support in Adabas

Tape Usage

The Adabas utilities can read from and write to tape devices directly. In order to make use of this ability, the corresponding environment variable must be set to point to the tape device-name. The utilities use their own tape format, which is not compatible with other tape handling utilities.

The environment variable must point to the name of the tape device before the utility is started. If, for example, you want to use the backup utility ADABCK on an HP-UX machine, you should enter the following:

C Shell:

```
setenv BCK001 /dev/rmt/0m
```

Bourne Shell, Korn Shell:

```
BCK001 = /dev/rmt/0m
export BCK001
```

If, for example, you want to use the backup utility ADABCK on Windows to write to a tape device \\.\TAPE0, you should enter the following:

Windows:

```
set BCK001=\\.\TAPE0
```

The device name of a tape drive has the form \\.\TAPE<tape id>, where <tape id> is the number of the tape device starting from zero.

Multiple-Tape Support

It may be necessary for very large files to be spread over two or more tapes. If a tape gets full, the utility informs the user of the situation and prompts for input. After changing the tape, the user must enter the device name again in order to continue.

Example:

```
%ADABCK-I-TDEVFU, end of tape reached, rewinding
                  Insert new tape and enter device name, [/dev/rmt/0m]  (UNIX)
                  Insert new tape and enter device name, [\\.\TAPE0]  (Windows)
```

A similar method is used when multiple tapes are read. The utility checks to see whether the tapes are mounted in the correct sequence. If the tapes are inserted in the wrong sequence, the utility issues an error message and prompts for input.

Example:

```
%ADABCK-F-ILLSUB, illegal subsequent tape detected,
                  present 1, expected 3
```

In order to continue, the user must mount the correct tape and enter the device name (the name of the tape device does not have to be the same as the first). Standard file names or raw sections may not be entered.

Automatic Tape Change

In order to support automatic tape change without prompting, you can set multiple tape devices within one environment variable. This means, for example, that if the first tape selected gets full, Adabas will automatically switch to the second etc. This mechanism is supported for both reading and writing tapes.

Example (C shell):

```
setenv  BCK001  "/dev/rmt/0m  /dev/rmt/1m"
setenv  RECPLG  "/dev/rmt/0m  /dev/rmt/1m"
```

or

```
setenv BCK001 /dev/rmt/0m
setenv BCK0012 /dev/rmt/1m
setenv RECPLG /dev/rmt/0m
setenv RECPLG2 /dev/rmt/1m
```

Example (Windows):

```
set BCK001=\\.\TAPE0;\\.\TAPE1
set RECPLG=\\.\TAPE0;\\.\TAPE1
```

A large database is to be dumped onto the two tapes selected. If two tapes are not enough, Adabas will prompt for a third one.

Calling Utilities

There is one important restriction that applies to using the utilities with tapes: input redirection is not allowed if more than one tape is required and the automatic tape feature described above is not used. In this case, the utility aborts with the following message:

```
%ADABCK-E-STNOTPO, subsequent tapes not possible with input redirection
```

The following modes will work successfully:

Interactive (UNIX)

```
adabck <RETURN>
dbid=5, dump=4 <RETURN>
<CTRL/D>
```

Interactive (Windows)

```
adabck <RETURN>
dbid=5, dump=4 <RETURN>
<CTRL/Z><RETURN>
```

Interactive, command line

```
adabck dbid=5 dump=4
```

Batch job (UNIX)

```
adabck dbid=5 dump=4 &
```

The utility process will be stopped if a subsequent tape is required. In this case, the user can bring the job into the foreground in order to enter a valid device name and continue processing.

Multiple Files on a Single Tape

Adabas supports multiple files on a single tape, provided that this feature is supported by the operating system in question (for example, via the mt interface with the no-rewind tape). In order to use this capability, skip to the desired position on the tape and then execute the utility.

Example:

```
mt fsf 1
adabck db=5 dump=3
```

The first file on the tape is skipped, and the database is dumped to tape as the second file.

Unsupported Tape Formats

Sometimes, external data (such as uncompressed data) may reside on tapes with a tape format not supported by Adabas, and this results in an I/O error from Adabas. Such tapes can be read using the UNIX command dd to read the file into a named pipe. The Adabas utility can then read from the named pipe.

Optimization of ADAMUP and ADAINV Execution

In order to create the index, the utilities ADAINV and ADAMUP must sort the descriptor values. The efficiency of the sort execution can be impacted by the size of utility parameter LWP, which specifies the amount of memory that can be used additionally to the minimum memory allocated for the utilities. In the following, the impact of the size of the LWP parameter for the execution of the sort operations is described.

- Depending on the size of the LWP parameter, an area in memory is provided for an in-memory sort. If LWP is large enough, all descriptor values can be sorted in memory, and the index can be created, and no SORT containers are required. The required value for an in-memory sort is displayed by the ADAMUP and ADAINV SUMMARY functions.
- If the area is too small for an in-memory sort, the data to be sorted are divided in subsets that can be sorted in memory. The number of subsets depends on the size of the LWP parameter: the number of subsets can be reduced by increasing the size of LWP. After the in-memory sort of these subsets, the sorted subsets are written to the SORT container.
- Once all of the subsets have been sorted, the sorted subsets must be merged. However, it is only possible to merge a limited number of subsets in a single pass. If the number of subsets does not exceed the limit, all descriptor values are sorted after the merge, and the index can be created. The required LWP value for a single- pass merge and the required size of the SORT container are displayed by the ADAMUP and ADAINV SUMMARY functions.
- If the number of subsets is higher, the merged subsets are written to disk again, and another merge pass is required; this also means that the SORT container must have be double the size

compared to the size required if a single merge pass is sufficient. The required size of the SORT container is displayed by the ADAMUP and ADAINV SUMMARY functions.

- The required size of the SORT container depends only on the number of merge passes, but not on the exact value of the LWP parameter: If the LWP parameter is large enough for a single-pass merge, you need half the amount of disk space that is required for a merge in two or more passes.
- An in-memory sort is not necessarily faster than a sort with one or more additional merge passes. The reason for this is that the sort algorithm randomly accesses the provided memory. With LWP=0, you usually don't have many memory-cache misses, but with a larger LWP parameter, you have a lot. This can increase the sort time more than the time required for merging the sorted subsets. Merging the sorted subsets doesn't cause many memory cache misses, and also the negative impact of the I/Os for the sort container is limited, since the I/Os are performed asynchronously, and it may happen that data are still in the file system cache for the sort container.
- The performance of the different system components may vary greatly between different computers; therefore, the recommendation is to try the different alternatives if you perform ADAMUP or ADAINV regularly. If you don't perform these utilities often, it may not be worth the effort.
- Performing ADAMUP SUMMARY is very fast, because the descriptor summary is already created when the MUPDTA and MUPDVT files are created, and is stored in the MUPDTA file. ADAINV SUMMARY, on the oither hand, must read all DATA blocks of the file to create the descriptor summary, and, therefore, is relatively slow.
- The SUMMARY functions also calculate the recommended TEMP sizes: the utilities can process only one descriptor at a time, but they always see the values of all descriptors. In order to prevent ADAINV having to reread the DATA blocks, the values for the other descriptors are stored on TEMP, grouped by descriptor. It should be possible to store at least the largest descriptor; ideally TEMP should be large enough to process all descriptors in one pass. Because ADAINV does not know the descriptor sizes in advance, the TEMP size required for one pass are relatively large; therefore ADAINV SUMMARY also displays the TEMP size required for processing the descriptors in two passes.
- The values computed by the SUMMARY functions are not the exact minimum values; the values are rounded up, i.e. these values are sufficient, but smaller values may be also sufficient.

Synchronization Between Nucleus and Utilities

The Utility Communication Block (UCB)

Most utilities can write to the database containers and can be performed both in online mode, i.e. when the nucleus is active, and in offline mode, i.e. when the nucleus is not active. This means that a synchronization mechanism is required for the nucleus and the utilities, independent of an active nucleus, in order to guarantee the integrity of the database.

For this purpose, each Adabas database contains a Utility Communication Block (UCB). The UCB is stored in RABN 3 of the ASSO container. Each time that utilities need to have read or write access for one or more files, corresponding UCB entries must first be created that contains the following information:

- The file number(s) of the Adabas file(s) to be processed;
- The access mode (read or write).

In addition, the access to some global data structures in the database containers must be synchronized between the nucleus and the utilities.

In online mode, the synchronization between utilities and nucleus is always done via the nucleus: The utilities issue special Adabas commands (U calls), and the nucleus updates the UCB and other global database blocks.

In offline mode, the utility itself updates the UCB and the other global database blocks; in order to avoid a situation in which two utility processes concurrently update these blocks, the access is protected via a semaphore. The semaphore protects only the access to the global data structures; the access to the file-specific data is protected via the UCB.

When a utility terminates, it normally removes its UCB entry again, but if a utility terminates abnormally, it can happen that the UCB entry for the utility is not removed. In this case, it is necessary to remove the UCB manually using ADADBM RESET=UCB.

Note: There are some read-only utilities that do not synchronize with the nucleus and other utilities (ADAREP and ADAPRI). This has the advantage that these utilities can always be executed, with the risk, that in some rare cases, it may happen that the utility terminates abnormally.

Databases in Read-Only Mode

A database can also be in read-only mode, either if the containers are write-protected, or via specifying the nucleus parameter OPTION=READONLY. For a database in read-only mode, it is not possible to write the UCB; nevertheless a synchronization is necessary because the read-only mode may be caused only by the nucleus parameter OPTION=READONLY, and the containers are writable. The nucleus and utilities access a shared memory where a General Control Block (GCB) is stored; this is a shared memory that is created by the first nucleus or utility process started, and removed again when the last nucleus and utility process using it, is terminated. This shared memory is used to mark the database as read-only. This is done when the shared memory is created, and either the containers are read-only, or the nucleus is started with OPTION=READONLY.

The following rules apply:

- If the shared memory is marked as read-only, it is only possible to start the nucleus in read-only mode, or to start utility functions that only read the database containers. A writing nucleus or utility can be started again only after the shared memory has been removed again;
- If the shared memory is not marked as read-only, it is not possible to start the nucleus in read-only mode, because it is possible that the utility that created the shared memory modifies the database, and a read-only nucleus assumes that this does not happen.

8 Loading And Unloading Data

Introduction

There are several ways of loading data into a database:

- Using the utility ADAMUP: you can load compressed data that was generated by ADACMP, ADAMUP or ADAULD.
- Using the utility ADABCK: you can restore a file or database from a backup that was created by ADABCK.
- Using the IMPORT function of the utility ADAORD: you can import a file that was created by the EXPORT function of ADAORD.

Compressed Data

Data that is loaded into the database by the utility ADAMUP must be input to ADAMUP in a special, compressed data format. Compressed data can be generated by the utility ADACMP, which converts uncompressed data, as described in the section Uncompressed Data Format, to the necessary format. Compressed data can also be generated by the utilities ADAMUP and ADAULD, when existing data is unloaded from the database.

This is a very flexible data format. You can load copies of a compressed data file into several different databases if required, or as several copies of the same file into a single database. You can also load just a subset of the records into a file.

A disadvantage of this file format is that when the data is loaded into the database, ADABAS has to build a sort index for each file. For large files, this can require large amounts of CPU time, and SORT and TEMP container files are required.

Backup Data

Backup data is generated by the utility ADABCK. Such data can be used to build a long-term data archive, and can also be used for restoring files to databases, or to restore whole databases.

The backup and restore operations are faster than the other methods of saving and restoring data. However, you must always copy backup files back to the database from which they originated. Also, you cannot copy the files back with different file numbers.

Import/Export Data

Data that was exported from a database using the ADAORD EXPORT function can be imported to a database using the ADAORD IMPORT function. The exported data is very similar to the compressed data format described above, but the main difference is that the index information of the exported files is also exported. This means that when data is subsequently imported, the index does not have to be rebuilt, so the load procedure is much faster than the corresponding operation for ADAMUP. Also, SORT and TEMP container files are not required.

Like compressed data, this is a flexible data format. You can load copies of a compressed data file into several different databases if required, or as several copies of the same file into a single database. However, because the index information is stored in the export file, you cannot import just a subset of the records into a file in the database.

Copying Data to other Hardware Architecture

The situation may occur in which you want to copy data from one Adabas database to another database on a computer with a different hardware architecture, for example from a UNIX platform to a Windows platform.

You can use the utility ADABCK (Version 6.4 and higher) for this purpose - you can restore a backup created on one hardware architecture into a database on a computer with another architecture.

Notes:

1. This is not possible with backups created with Adabas versions < 6.3.
2. Copying data in this way from or to mainframes is not supported.
3. Only ADABCK can process input files created on a different hardware architecture; the utilities ADAMUP and ADAORD are not able to do so.
4. Alternatively, you can use the old way of copying data to another hardware architecture that was required with previous Adabas versions: unload the file with the utility ADAULD, decompress the file with the utility ADADCU, compress the file with the utility ADACMP for the new architecture, load the file with ADAMUP. This may be useful if you also want to change the block sizes or the FDT of a file.

Uncompressed Data Format

This section describes the format of data records that are input to the utility ADACMP and output from the utility ADADCU. This format is called uncompressed data format (also called raw data format). The utility ADACMP reads in data in this format and compresses it for subsequent input to the mass update utility ADAMUP. The utility ADADCU performs the opposite operation: it takes compressed data that was generated by either ADACMP and decompresses it. Note that compressed data can also be generated by the utilities ADAMUP and ADAULD when data is deleted or unloaded from a database.

Unless otherwise indicated, the data formats described apply to both the input data for ADACMP and the output data for ADADCU.

Syntax of Uncompressed Data Records

Uncompressed data records are a sequence of the following syntax elements:

```
format_buffer_element
field_references
```

The syntax elements, except field_references, are the same as the format buffer elements described in Command Reference, Calling Adabas, Format and Record Buffers. Note that here a format buffer element is nX, a literal, or a field definition including the length and format specifications, if they exist. The difference is how the syntax elements are separated:

- The complete syntax element must be entered in one line.
- The syntax elements are separated by a comma or a newline.
- You can insert comments between the syntax elements: a semicolon indicates that the following characters, until end of line, are comments.
- You can insert *FDT* between the syntax elements; *FDT* must be entered in a new line. This indicates to the utility where the uncompressed data record syntax is specified that the FDT is to be displayed.

The following special considerations apply for format buffer elements specified in a decompressed record-structure specification:

- Edit masks are not allowed.
- N elements are not allowed.
- 1-N elements must be preceded directly in the same line by the corresponding C element. Unlike the format buffer for an update or store command, they are also allowed in format buffers for compression.

Example

Assume GB is a periodic group with the fields BA, BB and BC.

GBC,GB1-N	The number of occurrences of the periodic group and all occurrences of the periodic group are processed (GBC, BA1, BB1, BC1, BA2, BB2, BC2, ...).

- 1-N elements are not allowed for fields within a PE.

Examples for invalid specifications

GB2-GB4	Incorrect syntax.
GB4-2	Descending range.
GBC GB1-N	GBC and GB1-N must be specified in the same line.
GBC,BA1-N	name 1-N must not be specified for fields within a periodic group.

Syntax of field_references

```
name R [mu_pe_index] [, length ]
```

name must be the name of a LOB field. If the decompressed record specification contains field references, the decompressed record doesn't contain the LOB values themselves, but file names, where the LOB values are contained in the files.

length may be a number >= 0 or '*'. length=* is allowed only at the end of the record for a single LOB value. If length=0 is specified, a 2-byte inclusive length is put in front of the file name (analogous to LA fields). The default is 0.

Syntax of mu_pe_index

```
{ i [-j] | N} [ (m [-n] | N | 1-N) ]| 1 - N
```

If MU fields or fields in periodic groups are LOB fields, you can specify the MU or PE indices for field references the same way as you do for field values.

Notes:

1. The rules given above for the usage of 1-N as an MU-PE index for field values alsoapply for field references.
2. If the value of a LOB field is blank, the field reference is blank, too.

Example for a decompressed record structure specification

```
AA,AB           ; 2 fields specified in the same line
9X,'LITERAL'    ; Compress: 16 bytes are ignored
                ; Decompress: "         LITERAL" in decompressed record
fdt             ; Display FDT before next field is specified
AT1-12C         ; Number of values of MU field in the first 12 elements of PE
                ; Only allowed for decompress
AT1-12(1-2),8,U ; Values of MU field in a PE
                ; Length is 8 bytes, Format is U
P1C,P11-N       ; Periodic group count and all groups
                ; Allowed for compress, too.
LMR1-4,20       ; File names of files containing values
```

Record Definition Examples

This section provides record definition examples. All the examples in this section refer to the sample ADABAS files in Appendix A of the Command Reference Manual.

Example 1: Defining elementary fields (standard length and format):

```
Syntax        : AA,5X,AB.

Record        : AA value(8 bytes alphanumeric)
                5 spaces
                AB value(2 bytes packed)
```

Example 2: Defining elementary fields (length and format override):

```
Syntax        :  AA,5X,AB,3,U.

Record        :  AA value (8 bytes alphanumeric)
                 5 spaces
                 AB value (3 bytes unpacked)
```

Example 3: Processing a periodic group:

```
Syntax        : GB1.

Record        : BA1 value (1 byte binary)
                BB1 value (5 bytes packed)
                BC1 value (10 bytes alphanumeric)
```

Example 4: Processing the first two occurrences of periodic group GB:

```
Syntax        : GB1-2.

Record        : BA1 value (1 byte binary)
                BB1 value (5 bytes packed)        GB1
                BC1 value (10 bytes alphanumeric)

                BA2 value (1 byte binary)
                BB2 value (5 bytes packed)        GB2
                BC2 value (10 bytes alphanumeric)
```

Example 5: Processing the sixth value of the multiple-value field MF:

```
Syntax        : MF6.

Record        : MF value 6 (3 bytes alphanumeric)
```

Example 6: Processing the first two values of the multiple-value field MF:

```
Syntax        : MF01-02.

Record        : MF value 1 (3 bytes alphanumeric)
                MF value 2 (3 bytes alphanumeric)
```

Example 7: The highest occurrence number of the periodic group GC and the existing number of values for the multiple value field MF are processed:

```
Syntax        : GCC,MFC.

Record        : Highest occurrence count for GC (1 byte binary)
                Value count for MF (1 byte binary)
```

Output Record

The utility ADADCU returns the requested field values in the order specified by the record definition syntax. A value is returned in the standard length and format defined for the field, unless a length and/or format override was specified. If the value is a null value, it is returned in the format in effect for the field:

Format	Null Value
ALPHANUMERIC (A)	Blanks (ASCII: hex `20' or EBCDIC: hex `40')
BINARY (B)	Binary zeros
FIXED-POINT (F)	Binary zeros
FLOATING POINT (G)	Binary zeros
PACKED DECIMAL (P)	Packed decimal 0
UNPACKED DECIMAL (U)	Unpacked decimal 0, depending on the target architecture
UNICODE (W)	Blanks depending on WCHARSET specified

Note: For packed decimals, C is used as sign. For unpacked decimals, 3 is used as sign for target architecture ASCII, F for target architecture EBCDIC.

Adabas returns the number of bytes equal to the combined lengths (standard or overridden) of all requested fields.

Input Data Requirements for ADACMP

User data which is input to ADACMP must be contained in a sequential file. The fields in each record can be structured according to the field definition table provided or a subset of the file's fields.

User data which is input to ADACMP must be contained in a sequential file. There are four ways in which the records in the input file can be separated; please refer to the parameter RECORD_STRUCTURE in the chapter ADACMP in the Utilities Manual for more detailed information. The fields in each record must be structured according to the data definition statements provided.

If a user exit routine is used, the structure must agree with the data definitions after user exit processing. Any trailing information in an input record for which there is no corresponding data-definition statement will not be processed and will not be contained in the output produced by ADACMP.

Fields defined as UNPACKED must contain a valid sign value in the four high-order bits of the low-order byte. The sign must be in zoned-numeric format. ADABAS represents the signs in zoned format.

Fields defined as PACKED must contain a valid sign value in the four low-order bits of the low-order byte. Valid positive signs are A, C, E and F. Valid negative signs are B and D. ADABAS represents a positive value with a C and a negative value with a D.

Overpunch format is also supported. For detailed information, please refer to the VAX-11 Architecture Handbook, chapter Data Representation.

If the input file does not contain any records, a warning message is displayed and the utility aborts. However, a CMPDTA output file that contains the FDT information is created.

Multiple-Value Field Count

If the structure of the decompressed record is not described via the FIELDS parameter, please consider the following:

The values for a multiple value field must be preceded by a 1, 2 or 4 byte binary count, depending on the setting of the ADACMP parameter MUPE_C_L, to indicate the number of values of the multiple-value field in the record. The minimum number of values which may be specified is 1.

If the number of values is constant for each record, this number may be specified in the field definition table used to define the multiple-value field. In this case, the count byte in the input record must be omitted. This option is only enabled if the FDT keyword is used. FDTs that are read from the database always default to variable occurrence counts. These variable occurrence counts can be overwritten by using the FIELDS keyword.

Multiple fields within periodic groups must not be specified with an occurrence count when the periodic group has been specified with a variable occurrence count.

Example:

```
        01,PG,PE
         02,P1,4,A,NU
         02,PM,4,A,NU,MU(4)
                          ^
%ADACMP-E-FIXOCC, specification of occurrences not allowed at this position
```

The count provided by the user may be modified by ADACMP if the NU option is defined for the field. Null values are suppressed and the count field is modified accordingly.

Example :

```
Field Definition:  01,MF,4,A,MU,NU
```

Each record contains a variable number of values for MF.

```
Input Records       Before ADACMP          After ADACMP
Input Record 1
(3 values)          MF count = 3           MF count = 3
                    AAAA                   AAAA
                    BBBB                   BBBB
                    CCCC                   CCCC
Input Record 3
(3 values)          MF count = 3           MF count = 2
                    AAAA                   AAAA
                    <null value>           CCCC
                    CCCC
Input Record 4
(1 value)           MF count = 1           MF count = 0
                    <null value>
```

Example :

```
Field Definition:  01,MF,4,A,MU(3),NU
```

Each record contains 3 values for MF.

```
Input Records       Before ADACMP          After ADACMP
Input Record 1                             MF count = 3
                    AAAA                   AAAA
                    BBBB                   BBBB
                    CCCC                   CCCC
Input Record 2                             MF count = 2
                    AAAA                   AAAA
                    BBBB                   BBBB
                    <null value>
Input Record 3                             MF count = 2
                    AAAA                   AAAA
                    <null value>           CCCC
                    CCCC
Input Record 4                             MF count = 0
                    <null value>
                    <null value>
                    <null value>
```

Periodic Group Count

If the structure of the decompressed record is not described via the FIELDS parameter, please consider the following:

The first occurrence of a periodic group must be preceded by a 1, 2 or 4 byte binary count, depending of the ADACMP parameter MUPE_C_L, which indicates the number of occurrences of the periodic group in the record. The minimum number of occurrences which may be specified is 1.

If the number of occurrences is constant for each record, this number may be specified in the field definition table used to define the periodic group. In this case, the count byte in the input record must be omitted.

This option is only enabled when the FDT keyword is used. FDTs that are read from the database always default to variable occurrence counts. These variable occurrence counts can be overwritten by using the FIELD keyword.

The occurrence count provided may be modified by ADACMP only if all the fields in the periodic group are defined with the NU option. If all the fields in a given occurrence contain null values and there are no following occurrences which contain non-null values, the occurrence will be suppressed and the periodic group occurrence count will be modified accordingly.

Example (PE with NU):

```
Field Definitions:   01,GA,PE
                          02,A1,4,A,NU
                          02,A2,4,A,NU
```

The input records contain a variable number of occurrences for GA.

```
Input Records       Before ADACMP          After ADACMP
Input Record 1      GA count = 2           GA count = 2
                    GA (1st occ.)
                     A1 = AAAA              A1 = AAAA
                     A2 = BBBB              A2 = BBBB
                    GA (2nd occ.)
                     A1 = CCCC              A1 = CCCC
                     A2 = DDDD              A2 = DDDD
Input Record 2      GA count = 1           GA count = 0
                    GA (1st occ.)
                     A1 = <null value>      suppressed *
                     A2 = <null value>      suppressed *
Input Record 3      GA count = 3           GA count = 3
                    GA (1st occ.)
                     A1 = AAAA              A1 = AAAA
                     A2 = <null value>      A2 = suppressed
                    GA (2nd occ.)
                     A1 = BBBB              A1 = BBBB
                     A2 = <null value>      A2 = suppressed
```

```
                    GA (3rd occ.)
                     A1 = CCCC              A1 = CCCC
                     A2 = <null value>      A2 = suppressed
```

* but this is indicated by an empty field count of 2. Up to 63 consecutive empty fields are indicated by one appropriate empty field count.

Example (PE with NU):

```
Field Definitions:   01,GA,PE(3)
                           02,A1,4,A,NU
                           02,A2,4,A,NU
```

All input records contain 3 occurrences for GA.

```
Input Records       Before ADACMP          After ADACMP
Input Record 1
                    GA (1st occ.)          GA count = 3
                     A1 = AAAA              A1 = AAAA
                     A2 = <null value>      A2 suppressed
                    GA (2nd occ.)
                     A1 = BBBB              A1 = BBBB
                     A2 = <null value>      A2 suppressed
                    GA (3rd occ.)
                     A1 = CCCC              A1 = CCCC
                     A2 = <null value>      A2 suppressed
Input Record 2                             GA count = 2*
                    GA (1st occ.)
                     A1 = <null value>      A1 = suppressed
                     A2 = <null value>      A2 = suppressed
                    GA (2nd occ.)
                     A1 = BBBB              A1 = BBBB
                     A2 = <null value>      A2 = suppressed
                    GA (3rd occ.)
                     A1 = <null value>      A1 = suppressed
                     A2 = <null value>      A2 = suppressed
Input Record 3      All occ.               GA count = 0
                    contain                All occurrences
                    null value             are suppressed **
```

* The first occurrence is included in the count since occurrences follow which contain non-null values. The third occurrence is not included in the count since no occurrences follow which contain non-null values.

** but this is indicated by an empty field count of 2.

Example (PE without NU):

```
Field Definitions:   01,GA,PE(3)
                           02,A1,4,A
                           02,A2,4,A
```

All input records contain 3 occurrences for GA.

```
Input Records       Before ADACMP          After ADACMP
Input Record 1      GA (1st occ.)          GA count = 3
                     A1 = <null value>      A1 = <null value>
                     A2 = <null value>      A2 = <null value>
                    GA (2nd occ.)
                     A1 = <null value>      A1 = <null value>
                     A2 = <null value>      A2 = <null value>
                    GA (3rd occ.)
                     A1 = CCCC              A1 = CCCC
                     A2 = <null value>      A2 = <null value>
Input Record 2                             GA count = 3
                    GA (1st occ.)
                     A1 = <null value>      A1 = <null value>
                     A2 = AAAA              A2 = AAAA
                    GA (2nd occ.)
                     A1 = <null value>      A1 = <null value>
                     A2 = <null value>      A2 = <null value>
                    GA (3rd occ.)
                     A1 = <null value>      A1 = <null value>
                     A2 = <null value>      A2 = <null value>
```

Variable-Length Indicator

Each value of a variable-length field (length set to zero in the field definition) must be preceded by a length indicator (in binary format) which indicates the value length (including the length indicator).

The length of the length indicator is:

- 4 bytes, if the field has the L4 option
- 2 bytes, if the field has the LA option
- 1 byte, if the field has neither of these options

Example:

```
Field Definitions:

     01,AA,8,A,DE
     01,V1,0,A
     01,V2,0,A,LA
     01,V4,0,A,L4
```

Input records (high-order first)

```
"FIELD AA\x09FIELD V1\x00\x0aFIELD V2\x00\x00\x00\x0cFIELD V4"

"FIELD AA\x09FIELD V1\x07\xD2 (2000 data bytes)\x00\x00\x07\xD2 (2000 data bytes)"
```

Input records (low-order first)

```
"FIELD AA\x09FIELD V1\x0a\x00FIELD V2\x0c\x00\x00\x00FIELD V4"

"FIELD AA\x09FIELD V1\xD2\x07 (2000 data bytes)\xD2\x07\x00\x00 (2000 data bytes)"
```

NC Option Indicator

The values for fields with the NC option are defined without the indicator when the FDT is used to describe the input record

Example:

```
Field Definitions:

     01,AA,5,A,NC
     01,AB,5,A,NC

Input Record

Field AA     Field AB

(5 bytes)    (5 bytes)
```

If the input record contains values for the NC option, then either the NULL_VALUE parameter must be set, or the structure of the records must be described using the FIELDS parameter.

ADACMP Processing Considerations

Data Modifications

ADACMP modifies all input records as follows:

Fields defined with format U or P are checked to ensure that the field value is numeric and in the correct format.

If a value is null, it must contain characters which correspond to the format specified for the field:

Format	Null Value
ALPHANUMERIC (A)	Blanks (ASCII: hex `20' or EBCDIC: hex `40')
BINARY (B)	Binary zeros
FIXED-POINT (F)	Binary zeros
FLOATING POINT (G)	Binary zeros
PACKED DECIMAL (P)	Packed decimal 0
UNPACKED DECIMAL (U)	Unpacked decimal 0, depending on the source architecture
UNICODE (W)	Blanks depending on WCHARSET specified

For a packed or unpacked alphanumeric field, -0 is converted to +0

Any record which contains invalid data is written to the ADACMP error file and will not be written to the compressed file.

Data Compression

The value for each field is compressed (unless the FI option is specified) as follows:

- Trailing blanks are removed for fields defined with A format;
- Leading zeros are removed for fields defined with B, P or U format;
- If the field is defined with the NU option and the value is a null value, a one-byte indicator is stored. Hexadecimal `C1' indicates that one empty field follows, `C2' two, etc.;
- Empty fields located at the end of the record are not stored.

Example :

The following data definitions and corresponding values would be processed by ADACMP as shown in the following figure:

```
01,ID,4,B,DE      ; ID
01,BD,6,U,DE,NU   ; BIRTHDATE
01,SA,5,P         ; SALARY
01,DI,2,P,NU      ; DAYS ILL
01,FN,8,A,NU      ; FIRST_NAME
01,LN,9,A,NU      ; LAST_NAME
01,SE,1,A,FI      ; SEX
01,HO,7,A,NU      ; HOBBY
```

```
Field    Format  Before compression          After compression

ID         B     67 12 00 00                03  67 12

BD         U     31 36 30 35 35 39          07  31 36 30 35 35 39

SA         P     00 00 05 00 0C             04  05 00 0C

DI         P     00 0C                     )
                                           )C2 (two empty fields)
FN         A     20 20 20 20 20 20 20 20   )

LN         A     4E 41 4D 45 20 20 20 20 20  05  4E 41 4D 45

SE         A     4D                         4D

HO         A     20 20 20 20 20 20 20       C1 (one empty field)
```

ADAMUP Processing Considerations

When adding records to or deleting records from an ADABAS database file, entries have to be inserted/removed in the Address Converter (AC), Data Storage (DS) and in the index. The data storage space table (DSST) has to be modified accordingly.

Adding Records

ISN Assignment

If the USERISN option is set, the ISN given with the input data is used. If this ISN exceeds the current limit (MAXISN) for the file or has already been assigned to another record, ADAMUP terminates execution and returns an error message. As with an ADABAS N2 command, there is no automatic extension of the file's Address Converter. The file's first free ISN is set to a value that is one greater than the highest USERISN provided if there is a USERISN which is greater than or equal to the file's current first free ISN.

If the USERISN option is omitted or NOUSERISN is specified, the ISN of each record is assigned by ADAMUP. ISNs are assigned in ascending sequence. If ISN-reusage is enabled, ADAMUP first

scans the file's Address Converter for unused ISNs. Once all ISNs have been reused or if ISN-reusage is disabled, ADAMUP assigns new ISNs starting at the file's first free ISN. Whenever a new Address Converter block is required, it is taken from the extents that are currently available. When these blocks are exhausted, an automatic extension is carried out according to the rules described in this chapter. Processing continues if the extension is successful, otherwise ADAMUP terminates with an error message.

ISNs deleted by a mass delete that is running in parallel can be reused immediately for the records being added.

Finding Space In Data Storage

If DS-reusage has been enabled, ADAMUP scans the DSST for a DS RABN with sufficient space to store the current data record. One DSST RABN is scanned at a time, just as the ADABAS nucleus does, and the first free DS RABN is used if no space is found via the DSST. When a mass delete is run in parallel, the DS RABNs from which records are deleted are reused first. This is different to the procedure used by the ADABAS nucleus, but saves scanning the DSST and minimizes the number of I/Os to the Data Storage. This is because those RABNs have to be read and written by the delete routines in any case.

If DS-reusage is disabled, or if no space is found via the DSST, ADAMUP assigns a new DS block starting at the first free DS RABN.

Whenever new records are added to a Data Storage block, the padding factor specified for the file is considered. If a new Data Storage block is required, it is taken from the extents that are currently available. When these blocks are exhausted, an automatic extension is carried out. Processing continues if the extension is successful, otherwise ADAMUP terminates with an error message.

Deleting Records

In the first step, all input records on the file that contains the ISNs to be deleted are read and validated. If any invalid records are found, the line number and offset are reported, and ADAMUP terminates execution and returns an error status once the input file has been parsed completely.

At the end of this step, ADAMUP builds a table of the ISNs to be deleted in virtual memory. This table is used in the next steps when performing the updates required on the file's Address Converter, Data Storage and index. The space required for this table (one bit per entry) depends on the lowest and highest ISN specified on the input file. ADAMUP terminates execution and returns an error message if there is not sufficient space.

In the second step, the file's Address Converter is processed. Because the ISNs to be deleted are pre-sorted, the number of Address Converter IOs can be reduced to a minimum in this step.

The corresponding Address Converter entry of each ISN specified is retrieved. For unused ISNs, an entry is written to the error log and processing continues if NOT_PRESENT=IGNORE is specified (default), otherwise ADAMUP terminates and an error message is returned. For ISNs that are

used, the corresponding Data Storage RABN is put into the SORT and the Address Converter entry is deleted. Consecutive references to the same Data Storage RABN are skipped. Each Data Storage RABN put into the SORT is prefixed with the extent number to indicate its location in the File Control Block (FCB). This allows the next step to process the file's Data Storage according to the sequence in which the Data Storage extents were allocated.

At the end of this step, the first free ISN on the file is reset to the first ISN of the highest range of ISNs to be deleted, if ISN-reusage is enabled, and the highest ISN of the range of records to be deleted is identical to the last used ISN on the file.

In the third step, the file's Data Storage and Data Storage Space Table are processed. Because the Data Storage RABNs to be modified are now pre-sorted, the number of Data Storage and Data Storage Space Table IOs can be reduced to a minimum in this step.

The relevant Data Storage blocks are read using the values returned by the SORT. Within each block, the records identified by an ISN in the table of ISNs to be deleted are removed, the block is refilled with records to be added (when a mass add is run in parallel and DS reusage is enabled) and the Data Storage Space Table is modified accordingly. At the end of this step, the first free Data Storage RABN is reset to the start RABN of the last range of Data Storage RABNs from which all data were deleted, if DS reusage is enabled, and the end RABN is identical to the last used Data Storage RABN on the file.

Updating the Index

Once the Address Converter, Data Storage and Data Storage Space Table have been modified, ADAMUP copies the file's Normal Index (NI) to an intermediate file and resets the file's index extents. Index entries that correspond to deleted records are omitted in this step.

Loading the Normal and Main Index

In order to build the Normal Index and Main Index, the Descriptor Value Table (DVT) entries contained on the input file have to be read and sorted according to ascending descriptor values and ISNs. The output of this sort is merged with the Normal Index entries saved on the intermediate file, and is then used to build the new Normal Index and Main Index.

Descriptors defined with the unique option are checked to ensure that the new Normal Index contains only one ISN per descriptor value. If there is more than one ISN, the conflicting ISNs are written to the error log, the unique flag is reset in the FDT and processing continues if UQ_CONFLICT=RESET is specified. Otherwise ADAMUP terminates with an error message.

Besides sorting the descriptor values, reading the Descriptor Value Tables is very time-consuming as a result of the large number of I/Os to the sequential input file. Therefore, if there are many descriptors, ADAMUP attempts to minimize the number of passes required to read through the Descriptor Value Tables by using the information contained in the Descriptor Space Summary (DSS). During each pass through the Descriptor Value Tables, the values for one descriptor are directly given to the SORT. The values of additional descriptors, if they exist, are written to the

TEMP data set. The greater the number of descriptors using the TEMP in parallel during each pass, the faster this step will be. ADAMUP displays the total number of passes required at the end of the run.

All index blocks are filled in accordance with the padding factor specified when the file was loaded. Whenever a new index block is required, it is taken from the existing extents (which have been reset at the start of this step). If these blocks are exhausted, an automatic extension is carried out. Processing continues if the extension is successful, otherwise ADAMUP terminates with an error message.

Loading the Upper Index

Whereas the Normal Index and Main Index are organized on a descriptor-by-descriptor basis, the Upper Index, index level 3 and higher, contains all descriptors. In order to link in the new Main Index, an entry is made in the Upper Index for each new Main Index block. The whole Upper Index is rebuilt. The padding factor specified when loading the file is re-established. All pre-allocated blocks are used before additional blocks are allocated. If additional blocks are required, the procedure as described for Normal Index and Main Index loading is used.

Rejected Data

Any rejected data is written to the ADAMUP error file. The contents of this error file should be displayed using the ADAERR utility. Do not print the error file using the standard operating system print utilities, since the records contain unprintable characters.

Please refer to the ADAERR utility in the Utilities Manual for further information.

ADABCK Processing Considerations

The DUMP/EXU_DUMP Function

When dumping a complete database (DUMP=*), the database's global information and all loaded files are dumped to an ADABAS backup copy. Therefore, a database can be restored from a database backup copy. Single files contained in such ADABAS backup copies can also be restored.

Dumping only selected files allows a controlled backup of certain parts of a database in cases where backing up the complete database is unnecessary.

The DUMP/EXU_DUMP function may be used when the nucleus is active or inactive. If the nucleus is active during a DUMP, all updates are dumped to the backup copy.

The DUMP/EXU_DUMP function cannot be used when AUTORESTART is pending. Then first the nucleus has to be started to resolve the AUTORESTART pending situation.

When the DUMP is about to terminate, all transactions have to be synchronized on ET status. An active nucleus does this automatically on request of ADABCK. During synchronization, the nucleus will only schedule commands which

- enable ET users to attain ET status;
- complete any active update commands;
- are read/search commands.

If the nucleus terminates abnormally while a DUMP/EXU_DUMP is being executed, a message is sent to the database operator, requesting the nucleus to be started.

The nucleus may come up while the DUMP function is running. In this case, the nucleus and the DUMP function will synchronize with each other. The nucleus can be shut down with ADAOPR CANCEL while the DUMP function is active. If the nucleus terminates abnormally, ADABCK displays a message requesting the nucleus to be started. Then it waits until the nucleus performs its autorestart, after which it terminates normally.

Parallel Backups

Sometimes it can be useful to dump single files in parallel using multiple ADABCK jobs. This is generally possible with EXU_DUMP, but if the nucleus is active, only one DUMP function is permitted.

Note: Parallel backups are not supported on Windows platforms.

The RESTORE/OVERLAY Function

A backup copy can be used to restore/overlay either selected files or a database if single files or the database's global information is corrupted.

A backup copy that is created with ADABCK DUMP=* can be used to restore a complete database.

A backup copy that is created with ADABCK DUMP=(list of files), for example ADABCK DUMP=(2, 5-8, 100-130), contains only selected files from the database. This kind of backup will normally be used if you want to restore the specified files. You can also use such a backup to perform a complete database restore or overlay with ADABCK RESTORE=* or ADABCK OVERLAY=*. Note, however, that if you do this, only the files that were backed up by the ADABCK DUMP command will be restored to the database and all other files will be deleted. In this case, ADABCK will output the message GCBAFL to indicate that the total number of files in the database has changed as a result of the restore or overlay operation.

When restoring/overlaying files, the nucleus may be either active or inactive. A check is made that all of the RABNs required by the files to be restored/overlaid are available. If all RABNs are available, the file is restored to the same position as before. If one or more of the required RABNs are not available in the database, a completely new set of RABNs will be allocated.

When restoring/overlaying files, the nucleus may be either active or inactive. The RABNs required by the files to be restored/overlaid must be available.

The nucleus may not be active when restoring/overlaying a database, since exclusive control over the database container files is required.

When restoring/overlaying a complete database, the underlying database may be larger, containing more blocks or more containers than the backup save set. However, the block sizes covered by the save set must be identical. The unused blocks from the underlying database will be kept and their space will be returned to the free space table.

When restoring/overlaying files, the underlying database can be smaller or larger than the backup copy.

When restoring/overlaying files, ADABCK tries by default to restore the blocks to the original block numbers. If this space is not available because it is occupied by another file, the file will be completely restored to other block numbers, and an attempt is made to combine several file extents into one.

Parallel Restores

Sometimes it can be useful to restore single files in parallel using multiple ADABCK jobs. This is possible with both the RESTORE and the OVERLAY function, regardless of whether the nucleus is active or inactive.

Security File Considerations

When restoring/overlaying the security file, only the passwords and the associated permission levels are re-established; the protection levels of the files loaded are not re-established. Therefore, if the file is restored to a newly-formatted database, the protection levels have to be reenabled using the ADASCR security utility.

The protection levels of all files are only re-established if a database is restored/overlaid.

ADABCK Restart Considerations

ADABCK has no restart capability. An abnormally-terminated ADABCK execution must be rerun from the beginning.

An interrupted RESTORE/OVERLAY of one or more files will result in lost RABNs which can be recovered by executing the RECOVER function of the utility ADADBM. An interrupted RESTORE/OVERLAY of a database results in a database that cannot be accessed.

ADAORD Processing Considerations

Exporting Files

When exporting one or more files, ADAORD copies the content of each file's Data Storage together with the information required to re-establish its index to a sequential output file (ORDEXP). Exporting a file's data records is identical to unloading them, and ADAORD supports the same processing sequences as the ADAULD utility. There are, however, differences in the way in which the information required to re-establish the file's index is provided. ADAORD does not generate descriptor value table (DVT) entries based on the data records (like ADAULD), but rather retrieves and exports the file's inverted lists. This requires access to a valid index and results in additional I/Os on the one hand, while saving CPU time on the other.

All files to be processed are written to a single sequential output file (ORDEXP) in ascending file number sequence. Splitting the export into separate runs and thus creating several versions of the sequential output file should be considered if non-default allocation quantities or placements are to be used when subsequently re-importing a file. If non-default values and placements are used, each file requires a separate run, and splitting the export procedure helps prevent lengthy and time-consuming positioning during the re-import process.

Importing Files

When importing one or more files, ADAORD retrieves the information contained on the sequential input file (ORDEXP) to re-establish each file's Data Storage, Address Converter and index. Importing a file's data records and building the Address Converter is identical to loading them using the utility ADAMUP (with the USERISN option). However, the process of building the file's index is faster in ADAORD because the descriptor values and ISNs are provided in their correct sequence. This eliminates the necessity of sorting (and of using the SORT and TEMP files) and more than compensates for the additional expenditure that results from reading through the index during the EXPORT phase.

The format of the sequential input file (ORDEXP) is independent of any database device types. Therefore, the process of exporting and then re-importing can be used to migrate files between databases that reside on different device types.

When importing the security file, only the passwords and the associated permission levels are reestablished; the protection levels of the files imported are not reestablished. Therefore, if a file is imported to a newly-formatted database, the protection levels have to be re-enabled using the utility ADASCR (refer to the Utilities Manual for further information).

When importing the security file, only the passwords and the associated permission levels are re-established. If the files being imported are not already security protected in the database, their protection levels saved on the backup copy are restored.

Allocating Space

When importing a file, both the placement and initial allocation quantities can be controlled by the user or left to ADAORD.

Unless positioning is forced by the specification of a start RABN or the KEEP_LAYOUT option, ADAORD will use the following sequence for the initial allocation of a file's extents: Address Converter (AC), Upper Index (UI), Normal Index (NI) and Data Storage (DS).

Unless positioning is forced by the specification of a start RABN, ADAORD will use the following sequence for the initial allocation of a file's extents: Address Converter (AC), Upper Index (UI), Normal Index (NI) and Data Storage (DS).

This allows the two extent types with the highest probability of being exhausted (NI and DS) to be extended without breaking into another extent.

If the number of blocks or cylinders to be allocated is omitted and the KEEP_LAYOUT option has not been specified, ADAORD calculates the allocation quantity as follows:

ALQN = ALQO * (1 + (PFACN - PFACO) / 100)

where:

ALQN	New allocation quantity in blocks
ALQO	Old allocation quantity in blocks
PFACN	New padding factor
PFACO	Old padding factor

By default, the initial and all subsequent allocations will be made using a contiguous-best-try method. Specifying the CONTIGUOUS parameter ensures that only the first logical extent of each type specified is used, with the risk of ADAORD aborting if insufficient contiguous space is available.

If the number of blocks or megabytes to be allocated is omitted, ADAORD calculates the allocation quantity as follows:

ALQN = ALQO * (100 - PFACO) / (100 - PFACN)

where:

ALQN	New allocation quantity in blocks or megabytes
ALQO	Old allocation quantity in blocks or megabytes
PFACN	New padding factor
PFACO	Old padding factor

By default, the initial and all subsequent allocations will be made using a contiguous-best-try method.

ISN Assignment

The ISN provided with each data record (and also contained in the inverted lists) is used. ADAORD will terminate execution and return an error message if the limit (MAXISN) for a file has been decreased to a value less than the file's first free ISN and an ISN that exceeds the new limit is found. The file's new first free ISN is set to a value one greater than the highest ISN found in the data records.

In order to change the ISN assignment, the file has to be unloaded using ADAULD and then re-loaded using ADAMUP.

Reordering a Database

This function consists of implicit EXPORT and IMPORT functions.

When reordering at the database level, all of the files in the database have to be exported in the first step. A single version of ORDEXP will be created, independently of where it physically resides.

The second step consists of rearranging the database's FCB and FDT area and reallocating the DSST behind it.

The final step is to re-import the files. Each file is relocated, multiple logical extents are condensed into a single logical extent and the padding factors are reestablished.

The created sequential file (ORDEXP) will not be deleted at the end of this function.

When reordering at the file level and using a disk as intermediate storage, ADAORD minimizes temporary space requirements by creating/deleting the sequential file ORDEXP before/after processing each of the files specified. This leads to cycles in which an EXPORT is followed by an IMPORT function. This is possible because the files do not change their placement in the database and overwriting only occurs within a file's allocated areas. However, when the NODELETE option has been specified or the file ORDEXP resides on tape, ADAORD creates a single file and proceeds as described for reordering at the database level. On a tape, this eliminates problems that are the result of multiple versions of ORDEXP and also improves tape handling.

Repairing an Inconsistent Index

Because the new index is based on the content of the old index (and not on the file's data records), an index which is logically inconsistent cannot be repaired by exporting and re-importing the file. Furthermore, an index which is physically corrupted may cause ADAORD's EXPORT function to loop or terminate abnormally.

The index can only be repaired by either reinverting (using ADAINV) or unloading and reloading the file (using ADAULD and ADAMUP).

File Space Estimation

This section contains formulas for calculating the Associator and Data Storage space requirements for a file.

Getting a First Estimate

The following pages of this chapter describe how to get a reasonably accurate estimate of the disk space requirements for your file or database before you load the data. A simple way of getting a first approximation is to load a small amount of your data, for example 1%-2%, into the database, then run the ADAREP utility and check the figures output for "allocated" and "unused" blocks. Then extrapolate these figures to calculate how much space would be required for the full 100% of the data. This is the approach often used by experienced database administrators at customer sites to calculate space requirements.

Associator Space Estimation

The Associator space required for a file is the sum of the space requirements for the following Associator elements:

1. Normal Index

   The Normal Index is the lowest level of the index structure. It contains the inverted lists. Each inverted list is composed of a descriptor value and the list of ISNs of all the records that refer to this descriptor value.

2. Upper Index

   The Upper Index consists of the Main Index and the other upper index levels. The Main Index is the next-highest level of the index structure after the Normal Index. It is used to manage the Normal Index. Up to this level, each index block may contain entries for only one descriptor.

   The Upper Index (index levels 3 and higher) contains entries for all descriptors that are present. Level 3 is used to manage the Main Index. As long as there is more than one Upper Index block at the current level, more levels will be added, each level managing the level below.

3. Address Converter

   The Address Converter consists of a table of RABNs, each of which indicates the Data Storage location of the record identified by a given ISN.

Normal Index Space Estimation

The space required for the Normal Index depends on the number and the characteristics of the descriptors contained in the file.

An estimate of the Normal Index space required for each descriptor can be made using the formula:

NIBY = (IL * UV * MAXISN) + DV * (L + 2)

where

NIBY
Normal Index space requirement (in bytes).

UV
The average number of unique values in each record for the descriptor.
If the descriptor is not defined with the MU option, UV is equal to or less than 1.
If the descriptor is defined with the NU option, UV is equal to the average number of values per record minus the percentage of records containing a null value. For example, if the average number of values per record is 1 and 20 percent of the values are null, UV is equal to 1 - 0.2 = 0.8.

MAXISN
The number of records permitted for the file (see MAXISN parameter of the utility ADAFDU).

DV
The number of different values of the descriptor in the file.

L
The average length of each different value of the descriptor. If the descriptor is not defined with the FI option, L is equal to the average length. If the descriptor is defined with the FI option, L is equal to the standard length of the descriptor.

IL
IL ISN size of 2 or 4 bytes.

The factor IL*UV*MAXISN represents the space required to store the ISNs, and the DV*L factor represents the space required to store the descriptor values.

For descriptors with numerous duplicate values, the factor IL*UV*MAXISN is the important factor. For descriptors with a large proportion of unique values, DV*L is the important factor.

This is only valid if the data is loaded using the mass update utility ADAMUP or if the index is created with the inverted list utility ADAINV. If the data is loaded using S1 calls, twice as much space may be required (in the worst case), and the blocks are not filled completely. New values

must be added to a block in sort sequence. If there is not enough space available in a block, in index block is split.

Example 1: Calculating bytes

Descriptor AA has an average of 1 value in each record. There are 50 different values for AA in the file. There are no null values for AA. The average value length is 3 bytes. The MAXISN setting for the file is 20000, the ISN size is 2 bytes.

```
Field Definition: 01,AA,5,U,DE
NI = (2 * 1 * 20,000) + 50*(3 + 2)
NI = 40,000 + 250
NI = 40,250 bytes
```

Example 2: Calculating bytes

Descriptor BB has an average of 1 value in each record. There are 20000 different values for BB in the file. There are no null values for BB. The average value length is 10 bytes. The MAXISN setting for the file is 20000, the ISN size is 4 bytes.

```
Field Definition: 01,BB,15,A,DE
NI = (4 * 1 * 20,000) + 20,000*(10 + 2)
NI = 80,000 + 240,000
NI = 320,000 bytes
```

Example 3: Calculating bytes

Descriptor CC is a multiple-value field with an average of 10 values in each record. There are approximately 300 different values for CC in the file. The average value length is 4 bytes. There is an average of 3 null values in each record. The MAXISN setting for the file is 20000, the ISN size is 4 bytes.

```
Field Definition: 01,CC,12,A,DE,MU,NU
NI = (4 * 7 * 20,000) + 300*(4 + 2)
NI = 560,000 + 1,800
NI = 561,800 bytes
```

Example 4: Calculating bytes

Descriptor DD is a field within a periodic group. Each record has an average of 5 values for DD. There are 10 different values for DD in the file. Each record has an average of 3 null values. The MAXISN setting for the file is 20000. The average value length is 5 bytes, the ISN size is 2 bytes.

```
Field Definition: 01,PX
                  02,DD,8,A,NU
NI = (2 * 2 * 20,000) + 10*(5 + 2)
NI = 80,000 + 70
NI = 80,070 bytes
```

Once the number of bytes required for the Normal Index has been determined, an estimate of the number of blocks required can be made using the following formula:

NIBL = NIBY / (BL * (1 - p / 100) - 3)

where

NIBL
NI space requirement in blocks

NIBY
NI space requirement in bytes

BL
Associator block length

p
Associator block padding factor

The result of the division should be rounded up to the next integer.

Example

```
NI requirement in bytes = 60,250
Device type RA92
Associator block padding factor = 10 percent
NIBL = 60,250 / (2044 * (1 - 10 / 100) - 3)
NIBL = 32+ = 33 blocks
```

Example 5 : Calculating blocks

```
NI requirement in bytes = 60,250
Device type 2 KB
Associator block padding factor = 10 percent
NIBL = 60,250 / (2048 * (1 - 10 / 100))
NIBL = 32+ = 33 blocks
```

Upper Index Space Estimation

The Upper Index consists of the Main Index and other upper index levels. Each Normal Index representation in the Main Index consists of a 9 byte fixed part and the descriptor value. The Main Index space requirement for each descriptor may be calculated using the formula:

```
MIBY   = NIBL * (L + 9)
```

where

MIBY
Main Index space requirement (in bytes)

NIBL
Normal Index space requirement (in blocks)

L
The average length of each different value of the descriptor. If the descriptor is not defined with the FI option, L is equal to the average length. If the descriptor is defined with the FI option, L is equal to the standard length of the descriptor. For fields with format A and W, the length of truncated descriptor values must be considered; the descriptor values are truncated at the first byte where they differ from the previous descriptor value.

Example 1: Calculating bytes

```
NI Block Requirement = 45 blocks
MI   = 45 * (3 + 9)
MI   = 540 bytes
```

The following formula may be used to convert the Main Index byte requirement to blocks:

```
MIBL = MIBY / (BL * (1 - P/ 100))
```

where

MIBL
Main Index space requirement (in blocks)

MIBY
Main Index space requirement (in bytes)

BL
Associator block length

p
Associator block padding factor

The result of the division is rounded up to the next integer.

Example 2: Calculating blocks

```
MI byte requirement = 540 bytes
Device type 2 KB
Associator block padding factor = 5 percent
MIBL = 540 / (2048 * (1 - 5 / 100))
MIBL = 0+ = 1 block
```

Overall Space Requirements

The highest upper index levels (level 3 and higher) contain entries for all descriptors of a file. The overall space requirements for the upper index can be obtained using the following formula:

```
UIBL = M * (1 + C + C**2 + C**3 + ... + C**13)
```

where

UIBL
Upper index space requirement in blocks

M
Sum of the Main Index space requirements for all descriptors of the file

C is given by the following formula:

```
C = (L + 13) / (BL * (1 - P/100))
```

where

L
Average length of all values of all descriptors of the file

BL
Associator block length

p
Associator block padding factor

Address Converter Space Estimation

The Address Converter for a file consists of a list of the relative ADABAS block numbers (RABNs), each of which represents the Data Storage block number in which a given record is stored. The block numbers are stored in ISN sequence, with the nth entry containing the Data Storage RABN for ISN n. Three bytes are required for each entry.

The space requirement for the Address Converter can be calculated using the formula:

```
AC = MAXISN * 3 / BL
```

where

AC
Address Converter space requirement (in blocks)

MAXISN
MAXISN setting for the file

BL
Associator block size

The result of the division is rounded up to the next integer.

Example:

```
MAXISN = 2,000,000
Device type 2 KB
AC     = 2,000,000 * 3 / 2048
AC     = 6,000,000 / 2048
AC     = 2929+ = 2930 blocks
```

Data Storage Space Estimation

The Data Storage space requirement can be estimated using the formula:

```
DS = N/(BW/L) + 1
```

where

DS
Data Storage space requirement (in blocks)

N
Number of records to be loaded into the file

B
Data Storage block size

p
Data Storage block padding factor

BW
Real amount of space used (minus padding factor) (B*(1-p/100))

L
Average record length

Example:

```
Number of records = 1,000,000
Average compressed record length = 50
Device type = 4 KB
Data Storage block padding factor = 5 percent
BW = 4096 * (1 - 5/100) = 3891
DS = 1,000,000/(3891/50) + 1 = 12,988 blocks
```

9 User Exits And Hyperexits

This chapter contains an explanation of the user exits and hyperexits that are supported by Adabas.

User Exits Overview

A user exit is a user-written routine that enables the user to participate in the processing performed by the Adabas nucleus or Adabas utilities. It is enabled by nucleus or utility input parameters. The user-written routine is dynamically loaded at the startup of the nucleus or utility, and is called at predefined stages in the processing of the nucleus or utilities.

The routines should be written in the C programming language.

The user exit must be present as a dynamic shared library (UNIX) or as a dynamic link library (Windows). This means that a user exit or hyperexit has to be compiled and linked with the corresponding options. A shared library requires position-independent code, therefore the compiler must be called with the PIC option. A dynamic link library (DLL) must be compiled for multi-threading.

Adabas uses environment variables/logical names to locate user exits. See the section Creating and Defining User Exits and Hyperexits for details.

The following user exits are available for Adabas:

User Exit	Use
Nucleus user exit 1	user processing on a direct Adabas call before it is processed by the nucleus (located using the environment variable/logical name ADAUEX_1)
Nucleus user exit 2	user processing at the close of a protection log file or command log file (located using the environment variable/logical name ADAUEX_2)
Nucleus user exit 4	user processing on a CLOG output record before it is written to the CLOG file (located using the environment variable/logical name ADAUEX_4)
ADACMP user exit 6	user processing on an ADACMP input record before it is compressed by ADACMP (located using the environment variable/logical name ADAUEX_6)
ADAULD user exit 7	user processing on a compressed ADABAS record before it is processed by ADAULD (located using the environment variable/logical name ADAUEX_7)
Nucleus user exit 11	user processing on a direct Adabas call before it is processed by the nucleus (located using the environment variable/logical name ADAUEX_11)
Nucleus user exit 14	user processing on a CLOG V6 output record before it is written to the CLOG file (located using the environment variable/logical name ADAUEX_14)
Nucleus user exit 21	set authentication credentials via the Adabas Server API Functions
ADALNK user exit 0	user processing before Adabas call execution (located using the environment variable/logical name LNKUEX_0)
ADALNK user exit 1	user processing after Adabas call execution (located using the environment variable LNKUEX_1)

User Exit	Use
ADALNKX user exit 0	user processing before Adabas call execution (located using the environment variable/logical name LNKUEX_ACBX_0)
ADALNKX user exit 1	user processing after Adabas call execution (located using the environment variable LNKUEX_ACBX_1)
XA user exit	See XA Support in this manual for details (located using the environment variable/logical name XAUEX_0)

At the startup of the nucleus, the user exit will be called first with an initialization call of the following form:

```
uex_X (0, UEX_INIT*)
```

where X is the number of the user exit and can have the value 1, 2 or 4.

The structure UEX_INIT (defined in adauex.h) is used for input/output values between ADABAS and the user exit. The input parameters are the database ID and the current version. The user exit must inform ADABAS if the code is re-entrant or non re-entrant by setting uex_type to either UEX_REENTRANT or UEX_N_REENTRANT. The code may be declared re-entrant if it uses only variables of storage class automatic. If the code is non re-entrant, the calls to the user exits are serialized by ADABAS, whereas if the code is re-entrant, the calls can be done in parallel.

User Exit Descriptions

Nucleus User Exit 1

Description

The ADABAS nucleus user exit 1 is a user exit that performs user processing on a direct ADABAS call. The routine is called when the processing of a command begins. The input parameters that are specified enable the user exit to change the parameters of the ADABAS call, or to reject the call so that the user who issued the call receives an ADABAS response 22 (invalid command).

The user exit is not allowed to change the command code or to change any of the buffer lengths that are specified in the ADABAS control block. Changing any of these values causes an ADABAS response 22 (invalid command) to be returned to the user who issued the command.

The nucleus user exit 1 is activated by setting USEREXITS=1 in ADANUC.

Input Parameters

```
Format: uex_1 ( pcb, pfb, prb, psb, pvb, pib, pcq )
```

pcb: Usage: Adabas control buffer
Type: unsigned char *
Access: read/write
Mechanism: by reference

For the structure of an ADABAS control block, see the Command Reference Manual.

pfb: Usage: Adabas format buffer
Type: unsigned char *
Access: read/write
Mechanism: by reference

If pcb is the 0-pointer, then pfb points to the UEX_INIT structure (at init call)

prb: Usage: Adabas record buffer
Type: unsigned char *
Access: read/write
Mechanism: by reference

psb: Usage: Adabas search buffer
Type: unsigned char *
Access: read/write
Mechanism: by reference

pvb: Usage: Adabas value buffer
Type: unsigned char *
Access: read/write
Mechanism: by reference

pib: Usage: Adabas ISN buffer
Type: unsigned char *
Access: read/write
Mechanism: by reference

pcq:	Usage:	Adabas command queue element
	Type:	struct cq_entry *
	Access:	read
	Mechanism:	by reference

For the structure of an Adabas command queue element, see the header file adauex.h.

Return Values

The user-exit return value is essentially an Adabas response code.

ADA_NORMAL:	Success. The nucleus checks whether illegal changes have been made in the control block. If no illegal changes are detected, the call is processed.
Else:	Failure. The Adabas call is rejected with a response 22 (invalid command).

Nucleus User Exit 2

Description

The ADABAS nucleus user exit 2 is a user exit that performs user processing on a close of the command log file (CLOG) or protection log file (PLOG). The user exit is called after the file is closed in the following situations:

- Nucleus shutdown (ADAOPR SHUTDOWN or CANCEL function)
- Forced PLOG/CLOG change (ADAOPR FEOF function)
- Automatic PLOG/CLOG change (new extent).
- New PLOG after online dump (ADABCK NEW_PLOG function)
- During an Autorestart

This user exit can also be used to archive PLOGs (e.g. with ADADEV).

The nucleus user exit 2 is activated by setting USEREXITS=2 in ADANUC.

Input Parameters

```
Format: uex_2 ( sess_num, dbid, env_var_cnt, logname, status )
```

sess_num:	Usage:	PLOG session number
	Type:	int *
	Access:	read
	Mechanism:	by reference

sess_num is the number of the nucleus session. A value of zero indicates that the current PLOG is closed. If the pointer is 0, then it is the INIT-call. In this case, the next parameter points to the UEX_INIT structure.

dbid:	Usage:	database identifier
	Type:	int *
	Access:	read
	Mechanism:	by reference

dbid is a pointer to the database identifier. If sess_num is the 0-pointer, then it points to the UEX_IMT structure (at init call).

env_var_cnt:	Usage:	environment variable counter
	Type:	int
	Access:	read
	Mechanism:	by value

env_var_cnt is the current environment variable counter that is used when using multiple environment variables. For the PLOG it will be 1 for NUCPLG, 2 for NUCPLG2, 3 for NUCPLG3 etc. The user exit is able to translate the environment variable in order to obtain the PLOG's path name.

logname:	Usage:	file name in section
	Type:	char *
	Access:	read
	Mechanism:	by reference

logname is the name of the PLOG/CLOG in the disk section, e.g. PLG.15 or PLG.11(3). Together with the path name of the section, logname can be used directly by ADADEV to save the PLOG/CLOG, even when closed.

status:	Usage:	calling status
	Type:	int
	Access:	read
	Mechanism:	by value

status indicates the time at which the user exit was called. The possible values are UEX2_SWITCH for a PLOG/CLOG change, UEX2_SHUTDOWN at the end of the nucleus session, and UEX2_AUTORESTART when the nucleus comes up following a crash during an Autorestart.

Return Values

The user-exit return value is essentially an ADABAS response code.

ADA_NORMAL:	Success.
Else:	Failure. The Adabas nucleus prints a message that contains the user-exit return code. The user exit is disabled and will not be called again.

Nucleus User Exit 4

Description

The Adabas nucleus user exit 4 is a user exit that performs user processing on a CLOG output record before it is written to the CLOG file. The user exit can shorten, extend or completely change the record. If the length or structure of the record is changed, the utility ADACLP, which is used to print CLOGs, may not be able to output the records that are created. A warning message is issued if a user exit was active when the CLOG records were written.

The nucleus user exit 4 is activated by setting USEREXITS=4 in ADANUC.

Input Parameters

```
Format: uex_4 ( pcl, pcq, pdsc)
```

pcl:	Usage:	CLOG output record
	Type:	struct cl_entry *
	Access:	read/write
	Mechanism:	by reference

For the structure of a CLOG record, see the CL_ENTRY structure definition contained in adauex.h.

The length of the record is contained in the first two bytes of the CLOG record. Bytes 3 and 4, which contain information for ADACLP, must not be changed. The nucleus restores these bytes after the call to the user-exit routine.

The original length of the user-exit input record must not be increased. Additional data may only be returned by the user exit using the additional descriptor. The length of the CLOG record returned plus the length of the additional buffer must not exceed 32763 bytes, which is the maximum record length that can be written to the CLOG. The Adabas nucleus then prints a message that contains the user-exit return code 0. The user exit is disabled and will not be called again.

If a CLOG record is to be suppressed, the user exit must set the first two bytes of the record, which contain the length, to zero.

pcq:	Usage:	Adabas command queue element
	Type:	struct cq_entry *
	Access:	read
	Mechanism:	by reference

If pcl is the 0-pointer, then it points to the UEX_INIT structure (at init call).

For the structure of an Adabas command queue element, see the description of user exit 1.

pdsc:	Usage:	buffer descriptor
	Type:	struct cl_dcs *
	Access:	write
	Mechanism:	by reference

pdsc is the address of a structure that describes a buffer which contains additional data for the CLOG record. The buffer is specified by length and a pointer. The information contained in the buffer is appended to the CLOG output record specified by the parameter pcl.

Return Values

The user-exit return value is essentially an ADABAS response code.

ADA_NORMAL:	Success.
Else:	Failure. The Adabas nucleus prints a message that contains the user-exit return code. The user exit is disabled and will not be called again.

ADACMP Utility User Exit 6

Description

The ADACMP utility user exit 6 is a user exit that performs user processing on an ADACMP input record before it is compressed. The user exit may change or skip records, insert additional records or terminate the compression. The action to be taken is indicated by user-exit return values that are interpreted by the ADACMP utility.

The utility user exit 6 is activated by setting the USEREXIT option in ADACMP.

Input Parameters

```
Format: uex_6 ( in_st , out_st )
```

in_st:	Usage:	user exit 6 input structure
	Type:	struct ue6_in *
	Access:	read
	Mechanism:	by reference

out_sc:	Usage:	user exit 6 output structure
	Type:	struct ue6_out *
	Access:	write
	Mechanism:	by reference

Return Values

The user-exit return value is returned in the output structure.

UE6_O_PROCESS:	Compress this record.
UE6_O_SKIP:	Skip this record without compressing it.
UE6_O_TERM:	Terminate compression of records immediately.
UE6_O_REPEAT:	Call user exit again with the same input record.

For the definition of these constants, the structure ue6_in and the structure ue6_out, see the include file adauex.h.

Other Information

The user exit routine must be written in C. The routine will be dynamically loaded.

ADACMP passes control to the user exit routine immediately after reading each input record. The user routine may modify, extend or shorten the record or may indicate to ADACMP that the record is not to be processed. One or more additional records created within the user exit may also be passed to ADACMP.

A pointer to an input parameter block and a pointer to an output parameter block are passed with each call (please see the header file adauex.h for more information). ADACMP provides the length and address of the input record area on each call. If an end-of-file condition is detected in the input file, ADACMP sets the input status to UE6_I_EOF and the input record length to 0. The user exit must place the address of the output area and the output record length into the output parameter block before returning to ADACMP. By default, ADACMP sets the input length and area and the output length and area to the same value. In order to leave a record unchanged, the user routine only has to execute a return instruction. If a record is not to be processed by ADACMP, the output status should be set to UE6_O_SKIP.

The user exit may indicate to ADACMP that control is to be returned to the user exit immediately upon ADACMP's processing of the current record (without reading the next record). This is done

by setting the output status to UE6_O_REPEAT before returning to ADACMP. This technique may be used to pass a record created within the user exit to ADACMP.

If the user exit returns UE6_O_TERM in the output status, no further records will be processed.

```
typedef struct ue6_in
{

  unsigned long             ue6_i_status;
#define UE6_I_NORMAL        1               /* standard call               */
#define UE6_I_EOF           2               /* call after EOF on input     */
#define UE6_I_REPEAT        3               /* repeat call on same record  */
                                            /* because of previous output  */
                                            /* status UE6_O_REPEAT

  unsigned long             ue6_i_len;      /* length of input record      */
  unsigned char*            ue6_I_ptr;      /* pointer to input record     */
} UE6_IN;
```

```
typedef struct ue6_out
{
  unsigned long             ue6_o_status;
#define UE6_O_PROCESS       1               /* process (compress) record   */
#define UE6_O_SKIP          2               /* skip this record            */
#define UE6_O_TERM          3               /* terminate compression       */
#define UE6_O_REPEAT        4               /* call again before reading   */
                                            /* next record from input file */

  unsigned long             ue6_o_len;      /* length of output record     */
  unsigned char*            ue6_o_rec;      /* pointer to output record    */
} UE6_OUT;
```

Example

```
#include <adabas.h>
#include <adauex.h>

#define  PERS_ID_OFFSET             0
#define  SEX_OFFSET                69
#define  FULL_ADDRESS_OFFSET       76
#define  ADDRESS_LENGTH            20
#define  CITY_COUNTRY_DISPLACEMENT 30
```

```
/*+
**      NAME:
**              uex_6 - adabas user exit 6 example
**
**      SYNOPSIS:
**              int     uex_6 ()
**
**      DESCRIPTION:
**              This USEREXIT requires uncompressed records
**              of the example file EMPLOYEES as input.
**
**              It changes the personnel-id for all employees
**              coming from Denmark (DK) and United States (USA).
**              The personnel-id is changed in the way that for
**              all female employees from Denmark the personnel-id
**              starts with "40", for all male employees with "41".
**              For all female employees in the United States the
**              personnel-id starts with "20", for all male
**              employees with "21".
**              Additionally all employees from Spain are rejected
**              and therefore those records are not compressed.
**
**      RETURN VALUES:
**              always 0
**
**      FUNCTIONS USED:
**              none.
**
-*/

#ifdef __STDC__

int uex_6 (struct ue6_in* ue6_in_ptr, struct ue6_out* ue6_out_ptr)

#else

int uex_6 (ue6_in_ptr, ue6_out_ptr)

UE6_IN  *ue6_in_ptr;
UE6_OUT *ue6_out_ptr;

#endif
{

  register unsigned char *country_ptr;
  register unsigned char *field_ptr;
  register unsigned char  mu_field_count;

  if (ue6_in_ptr->ue6_i_status == UE6_I_NORMAL)
  {

    /*
```

```
  ** calculate address of country in input record
  */

  field_ptr      =  ue6_in_ptr->ue6_i_ptr + FULL_ADDRESS_OFFSET;
  mu_field_count = *field_ptr;
  country_ptr    =  field_ptr + 1 + mu_field_count * ADDRESS_LENGTH +
                    CITY_COUNTRY_DISPLACEMENT;
```

```
  if (memcmp(country_ptr, "E  ", 3) == 0)
  {
    /*
    ** mark records of spanish employees to be skipped
    */

    ue6_out_ptr->ue6_o_status = UE6_O_SKIP;
  }
  else if ((memcmp(country_ptr, "USA", 3) == 0) ||
           (memcmp(country_ptr, "DK ", 3) == 0))
  {
    /*
    ** modify personnel id for employees from denmark and USA
    */

    field_ptr  = ue6_out_ptr->ue6_o_ptr + PERS_ID_OFFSET;

    if (memcmp(country_ptr, "USA", 3) == 0)
    {
      *field_ptr = '2';
    }
    else
    {
      *field_ptr = '4';
    }

    if (*(ue6_out_ptr->ue6_o_ptr + SEX_OFFSET) == 'F')
    {
      *(field_ptr + 1) = '0';
    }
    else
    {
      *(field_ptr + 1) = '1';
    }
  }
}
else
{

  /*
  ** signal termination of processing caused by EOF of input file
  */

  ue6_out_ptr->ue6_o_status = UE6_O_TERM;
```

```
  }
  return(0);
}
```

ADAULD Utility User Exit 7

Description

The Adabas ADAULD utility user exit 7 is a user exit that performs user processing on a compressed ADABAS record before it is unloaded by the ADAULD utility. The user exit may change or skip records or terminate unloading. The actions to be taken are indicated by user-exit return values that are interpreted by the ADAULD utility.

The utility user exit 7 is activated by setting the USEREXIT option in ADAULD.

Input Parameters

```
Format: uex_7 ( in_st, out_st )
```

in_st: Usage: user exit 7 input structure
Type: struct ue7_in *
Access: read/write
Mechanism: by reference

out_st: Usage: user exit 7 output structure
Type: struct ue7_out *
Access: write
Mechanism: by reference

Return Values

The user-exit return value is returned in the output-status field.

UE7_O_PROCESS:	Unload this record.
UE7_O_SKIP:	Skip this record without unloading it.
UE7_O_TERM:	Terminate unloading of records immediately.

For the definitions of these constants, the structure ue7_in and the structure ue7_out, see the include file adauex.h.

Parameter Block for User Exit 7

```
typedef struct ue7_in
{
  unsigned long          ue7_i_len;        /* length of input record         */
  unsigned char*         ue7_i_ptr;        /* pointer to input record        */
} UE7_IN;

typedef struct ue7_out
{

  unsigned long          ue7_o_status;
#define UE7_O_PROCESS    1                 /* process (unload) this record)  */
#define UE7_O_SKIP       2                 /* skip this record               */
#define UE7_O_TERM       3                 /* terminate unload               */
} UE7_OUT;
```

Example

```
#include <adabas.h>
#include <adauex.h>

#define FEMALE            'F'
#define MALE              'M'

#define REC_LNG            2     /* Record starts with two byte length field */
#define ISN_LNG            4     /* Next four bytes represent ISN            */
#define PERS_ID_LNG        9     /* Next nine bytes represent personnel id   */

#define FULL_NAME_OFFSET   (REC_LNG + ISN_LNG + PERS_ID_LNG)
                                          /* Offset to FULL-NAME group    */
#define EMPTY_FIELD_IND    0xC0           /* Indicator for empty NU-field */
```

```
/*+
**      NAME:
**              uex_7 - adabas user exit 7 example
**
**      SYNOPSIS:
**              int     uex_7 ()
**
**      DESCRIPTION:
**              This USEREXIT requires compressed records
**              of the example file EMPLOYEES as input.
**
**              It unloads all records of female employees and skips
**              all other records of male employees. If a value different
**              from 'F' or 'M' is found ADAULD is terminated.
```

```
**
**      RETURN VALUES:
**              always 0
**
**      FUNCTIONS USED:
**              none.
**
-*/

#ifdef __STDC__

int uex_7 (struct ue7_in* ue7_in_ptr, struct ue7_out* ue7_out_ptr)

#else

int uex_7 (ue7_in_ptr, ue7_out_ptr)

UE7_IN  *ue7_in_ptr;
UE7_OUT *ue7_out_ptr;

#endif
{
  register unsigned char *field_ptr;

  /*
  ** skip to first field of FULL-NAME group
  */

  field_ptr =  ue7_in_ptr->ue7_i_ptr + FULL_NAME_OFFSET;
```

```
  if (*field_ptr & EMPTY_FIELD_IND)
  {
    /*
    ** one empty NU-field (see FDT), skip to NAME-field
    */

    field_ptr += 1;
  }
  else
  {
    /*
    ** it's a length indicator, skip to NAME-field
    */

    field_ptr += *field_ptr;
  }

  field_ptr += *field_ptr;
                /* Add length byte of NAME-field, skip to MIDDLE-NAME-field */

  if (*field_ptr & EMPTY_FIELD_IND)
  {
```

```
    /*
    ** one empty NU-field (see FDT), skip to MARRIAGE-STATE-field
    */

    field_ptr += 1;
  }
  else
  {
    /*
    ** it's a length indicator, skip to MARRIAGE-STATE-field
    */

    field_ptr += *field_ptr;
  }

  field_ptr += 1;                                        /* Skip to SEX-field */

  if (field_ptr < (ue7_in_ptr->ue7_i_ptr + ue7_in_ptr->ue7_i_len))
  {
    if (*field_ptr == FEMALE)
    {
        return(0);                          /* Female employee, unload record */
    }
    else if (*field_ptr == MALE)
    {
      ue7_out_ptr->ue7_o_status = UE7_O_SKIP;
      return(0);                           /* Male employee, skip this record */
    }
  }

  /*
  ** something is wrong, terminate ADAULD
  */

  ue7_out_ptr->ue7_o_status = UE7_O_TERM;
  return(0);
}
```

Nucleus User Exit 11

Description

The Adabas nucleus user exit 11 is a user exit that performs user processing on a direct Adabas call. The routine is called when the processing of a command begins. The input parameters that are specified enable the user exit to change the parameters of the Adabas call, or to reject the call so that the user who issued the call receives an Adabas response 22 (invalid command).

The functionality of this user exit is the same as user exit 1, but it uses the new Adabas structures of version 6.

The user exit is not allowed to change the command code or to change any of the buffer lengths that are specified in the Adabas control block. Changing any of these values causes an Adabas response 22 (invalid command) to be returned to the user who issued the command.

The nucleus user exit 11 is activated by setting USEREXITS=11 in ADANUC.

Input Parameters

```
Format: uex_11 ( pacbx, pcb, pcq6, num_abd, patb_abd)
```

pacbx: Usage: New Adabas control block
Type: struct adacbx *
Access: read/write
Mechanism: by reference

For the structure of an Adabas control block, see the *Command Reference Manual*.

pcb: Usage: Old Adabas control block
Type: struct cb_par *
Access: read/write
Mechanism: by reference

For the structure of an Adabas control block, see the *Command Reference Manual*.

pcq6: Usage: Adabas command queue element V6
Type: struct v6_cq_entry *
Access: read
Mechanism: by reference

For the structure of an Adabas command queue element, see the header file adauex.h.

num_abd: Usage: Number of Adabas buffer descriptors (ABD)
Type: int
Access: read/write
Mechanism: by value

patb_abd:	Usage:	Pointer to ABD list
	Type:	char *
	Access:	read/write
	Mechanism:	by reference

For the structures of ACBX, ABD and ABD list, see the *Command Reference Manual*.

Return Values

The user-exit return value is essentially an Adabas response code.

ADA_NORMAL:	Success. The nucleus checks whether illegal changes have been made in the control block. If no illegal changes are detected, the call is processed.
Else:	Failure. The Adabas call is rejected with a response 22 (invalid command).

Nucleus User Exit 14

Description

The Adabas nucleus user exit 14 is a user exit that performs user processing on a CLOG output record of version 6 layout, before it is written to the CLOG file. The user exit can shorten, extend or completely change the record. If the length or structure of the record is changed, the utility PRILOGC, which is used to print the new type of CLOGs, may not be able to output the records that are created. A warning message is issued if a user exit was active when the CLOG records were written.

The nucleus user exit 14 is activated by setting USEREXITS=14 in ADANUC.

Input Parameters

```
Format: uex_14 ( pclx, pbuf, pcq6)
```

pclx:	Usage:	CLOG layout 6 output record
	Type:	struct clx_entry *
	Access:	read/write
	Mechanism:	by reference

For the structure of a CLOG layout 6 record, see the CLX_ENTRY structure definition contained in adauex.h

The length of the record is contained in the first two bytes of the CLOG record. Bytes 3 and 4, which contain information for PRILOGC, must not be changed. The nucleus restores these bytes after the call to the user-exit routine.

The original length of the user-exit input record must not be increased. Additional data may only be returned by the user exit using the additional descriptor. The length of the CLOG record returned plus the length of the additional buffer must not exceed 32763 bytes, which is the maximum record length that can be written to the CLOG. The Adabas nucleus then prints a message that contains the user-exit return code 0. The user exit is disabled and will not be called again.

If a CLOG record is to be suppressed, the user exit must set the first two bytes of the record, which contain the length, to zero.

If pclx is the 0-pointer, then it points to the UEX_INIT structure (at init call).

pbuf:	Usage:	buffer pointer
	Type:	char *
	Access:	write
	Mechanism:	by reference

pdsc is the address of a structure that describes a buffer which contains additional data for the CLOG record. The buffer is specified by length and a pointer. The information contained in the buffer is appended to the CLOG output record specified by the parameter pcl.

pcq6:	Usage:	Adabas command queue element V6
	Type:	struct v6_cq_entry *
	Access:	read
	Mechanism:	by reference

For the structure of an Adabas command queue element, see the header file adauex.h.

Return Values

The user-exit return value is essentially an Adabas response code.

ADA_NORMAL:	Success.
Else:	Failure. The Adabas nucleus prints a message that contains the user-exit return code. The user exit is disabled and will not be called again.

Nucleus User Exit 21

Description

The Adabas nucleus user exit 21 can be used to set authentication credentials via the ADABAS Server API Functions. The routine is called when the processing of a session begins.

This routine should be used as briefly as possible. It is intended for use during the transition period, until all applications use and support the Adabas Security authentication feature.

The input parameters that are specified enable the user exit to identify the calling application by analyzing the parameters of the Adabas call and set the appropriate credentials or to reject the call so that the user who issued the call receives an Adabas response 200 (security violation).

The nucleus user exit 21 is activated by setting USEREXITS=21 in ADANUC and the environment variable ADAUEX_21.

Input Parameters

```
Format: int uex_21 (uex21, uex_init)
```

uex21:	Usage:	Object Handle
	Type:	struct SECUEXStruct *
	Access:	read
	Mechanism:	by reference

For the structure of an Adabas control block, see the Command Reference Manual.

uex_init:	Usage:	Initialization Indicator
	Type:	struct uex_init *
	Access:	read/write
	Mechanism:	by reference

Return Values

The user-exit return value determines how the authentication processing proceeds.

SECUEX_SUCCESS:	Success. The authentication processing continues with the provided credentials.
SECUEX_FAILED:	Failure. The Adabas call is rejected with a response 200 (security violation).

Parameter Block for User Exit 21

```
struct SECUEXStruct {

    SECUEXPrivate * privatedata; /* For Internal Use Only */

    unsigned int  secdbid;  /* database id */
    FNR           secfnr;  /* file number */
    UQID          secuqid;  /* s-node, s-user, s-tid */

    int (*set_uid_psw)   (SECUEX * su, char * uid, char * psw);
    int (*get_acbx)      (SECUEX * su, ACBX * acbx);
    int (*is_natural)    (SECUEX * su);
    int (*is_sql_gateway)(SECUEX * su);
};
```

Functions / Methods

The user-exit provides the following functionality to enable the identification of the Adabas session:

- `*set_uid_psw` Set credentials for session.
- `*get_acbx` Retrieve the ACBX Control Block of the session.
- `*is_natural` Determine whether call was issued by a Natural application.
- `*is_sql_gateway` Determine whether call was issued by an SQL Gateway application

Function	`*set_uid_psw`	Set credentials for session
Parameter	`SECUEX * su`	Object Handle
	`char * uid`	Reference to User ID
	`char * psw`	Reference to Password
Return Value	`SECUEX_SUCCESS`	Function completed successfully
	`SECUEX_INVALID_PARAM`	Invalid or missing object handle
	`SECUEX_INVALID_INTERNAL`	Internal Error – unexpected values
	`SECUEX_INVALID_HEADER`	Internal Error – invalid security buffer
	`SECUEX_BUFFER_OVERFLOW`	Internal Error – security buffer overflow

Function	`*get_acbx`	Retrieve the ACBX Control Block of the session
Parameter	`SECUEX * su`	Object Handle
	`char * acbx`	Reference to struct ACBX
Return Value	`SECUEX_SUCCESS`	Function completed successfully
	`SECUEX_INVALID_PARAM`	Invalid or missing object handle
	`SECUEX_INVALID_INTERNAL`	Internal Error – unexpected values

For the structure of the Adabas control block, see the *Command Reference* documentation.

Function	*is_natural	Determine whether call was issued by a Natural application
Parameter	SECUEX * su	Object Handle
Return Value	SECUEX_TRUE	Calling application is Natural
	SECUEX_FALSE	Calling application is not Natural
	SECUEX_INVALID_PARAM	Invalid or missing object handle
	SECUEX_INVALID_INTERNAL	Internal Error – unexpected values

Function	*is_sql_gateway	Determine whether call was issued by an SQL Gateway application
Parameter	SECUEX * su	Object Handle
Return Value	SECUEX_TRUE	Calling application is SQL Gateway
	SECUEX_FALSE	Calling application is not SQL Gateway
	SECUEX_INVALID_PARAM	Invalid or missing object handle
	SECUEX_INVALID_INTERNAL	Internal Error – unexpected values

Example

```
#include <adauex.h>

/*
**  NAME:
**      uex_21 - Adabas Security Exit Example
**
**  SYNOPSIS:
**      int uex_21 ()
**      SECUEX   *   Pointer to SECUEX Data and Methods
**      UEX_INIT *   Pointer to UEX Initialization Mode
**
**  DESCRIPTION:
**      This user exit is called before an Adabas call is processed.
**
**      It provides the Security credentials,
**      which are to be used for authentication in this session.
**
**      Because no global variables are used, this exit flags
**      in init call to inform the Adabas nucleus that reentrant
**      code is being used.
**
**      To create a shared library that includes this user exit,
**      the makefile can be used.
**
**      Call:          make   -f makefile uex21   (Unix)
**                      nmake -f makefile uex21   (Windows)
**
**  RETURN VALUES:
**
**      SECUEX_SUCCESS
```

```
**      SECUEX_FAILED
*/

#ifdef __STDC__

int uex_21 (SECUEX    * uex21,
            UEX_INIT  * uex_init)

#else

int uex_21 (uex21,
            uex_init)
SECUEX    * uex21;
UEX_INIT  * uex_init;

#endif

{
  int rc;
  if (uex21 == 0)
  {
    /*
    **  User-Exit Initialization during Nucleus Startup
    **
    **  Indicate whether the user-exit is reentrant or not
    **  - UEX_REENTRANT
    **  - UEX_N_REENTRANT
    */
    uex_init->uex_type = UEX_REENTRANT;

    return ( SECUEX_SUCCESS );
  }

  else

  {
    /*
    **  AuthN Processing - during Session Initialization
    **
    **  Provide security credentials or not
    **  This can be based on the values in ACBX and UQID values
    **  which can be retrieved as needed (optional)
    **
    **  Note:
    **  Not providing security credentials
    **  will result a security violation "unable to authenticate".
    **
    */
    ACBX  acbx;
    char uid[32];
    char psw[32];
```

```
  /*
  ** Retrieve Session-Specific information
  */

  /* ACBX values */
  rc = uex21->get_acbx (uex21, &acbx);
  if (rc != SECUEX_SUCCESS)
  {
    return( SECUEX_FAILED );
  }

 /*
  ** Reject commands from users with names starting with 'h'
  */
  if ( uex21->secuqid.tid[0] == 'h' )
  {
    return( SECUEX_FAILED );
  }

  /*
  ** Reject acess to dbid=200
  ** where users names start with 'h'
  */
  if ((acbx.acbxdbid == 200 ) && (uex21->secuqid.tid[0] == 'h'))
  {
    return( SECUEX_FAILED );
  }

  /*
  ** Set Application-Specific credentials
  */
  strcpy (uid,"uexuid");
  strcpy (psw,"uexpsw");

  /* NATURAL */
  rc = uex21->is_natural (uex21);
  if (rc == SECUEX_TRUE)
  {
    strcpy (uid,"NATuid");
    strcpy (psw,"NATpsw");
  }

  /* SQL Gateway */
  rc = uex21->is_sql_gateway (uex21);
  if (rc == SECUEX_TRUE)
  {
    strcpy (uid,"SQLuid");
    strcpy (psw,"SQLpsw");
  }

  /*
  ** Set Security Credentials: userid, password
```

```
    */
    rc = uex21->set_uid_psw (uex21, uid, psw);
    if (rc != SECUEX_SUCCESS)
    {
      return( SECUEX_FAILED );
    }
  }

  return ( SECUEX_SUCCESS );       /* provided Security Credentials */
}
```

ADALNK-Specific User Exits

Overview

User exits for ADALNK can be used for any application that might either request control and/or want to modify Adabas parameters (e.g. the control block) during run time.

This user exit is called with the Adabas buffers as parameters (like the nucleus user exit 1). The user exit can be enabled before the Adabas call is passed to the nucleus, and after the Adabas call has been executed. The difference to the nucleus user exit 1 is that the ADALNK-specific user exit runs in the context of the user's process.

The ADALNK-specific user exits must be present as a shared library (UNIX) or as a dynamic link library (Windows). See the section Creating and Defining User Exits and Hyperexits for information about how to compile and link ADALNK-specific user exits.

Notes on Signal Handlers (UNIX)

If ADABAS calls are used in an application-defined signal handler and ADALNK user exits are established, it is possible that the user exit for the call from the signal handler is not started. This happens if the call from the signal handler occurs when another call of the user exit is currently active.

If the signal handler aborts a user application while an ADABAS call is active, the following situations can occur:

1. User exit 0 is already called but the ADABAS call is not passed to the nucleus.
2. The nucleus returns the results of the ADABAS call but user exit 1 is not called.

Notes on Exception Handlers (Windows)

If ADABAS calls are used in an application-defined exception handler and ADALNK user exits are established, it is possible that the user exit for the call from the exception handler is not started. This happens if the call from the exception handler occurs when another call of the user exit is currently active.

If the exception handler aborts a user application while an ADABAS call is active, the following situations can occur:

1. User exit 0 is already called but the Adabas call is not passed to the nucleus.
2. The nucleus returns the results of the Adabas call but user exit 1 is not called.

Input Parameters

```
Format:  int  lnkuex_{0 | 1} (cb,fb,rb,sb,vb,ib)
```

See the nucleus user exit 1 for a description of the input parameters.

Return Values

The user-exit return value is essentially an ADABAS response code.

ADA_NORMAL:	Success.
Else:	Failure. The response code will be placed into the Adabas control block. In the case of lnkuex_0, the call will return immediately to the user. If the call is an `RC' call, response code will be set to 0.

Creating a Link Between ADALNK-Specific User Exits and ADALNK

In order to establish a link between the equivalent user exits and ADALNK itself, the following environment variables have to be set:

- User exit before database access inside ADALNK

```
setenv LNKUEX_0 lnk_uex.sl (UNIX)
```

or

```
set LNKUEX_0=lnk_uex.dll (Windows)
```

Note: The default entry function name is 'lnkuex_0(...)'.

- User exit after database access inside ADALNK

```
setenv LNKUEX_1 lnk_uex.sl (UNIX)
```

or

```
set LNKUEX_1=lnk_uex.dll (Windows)
```

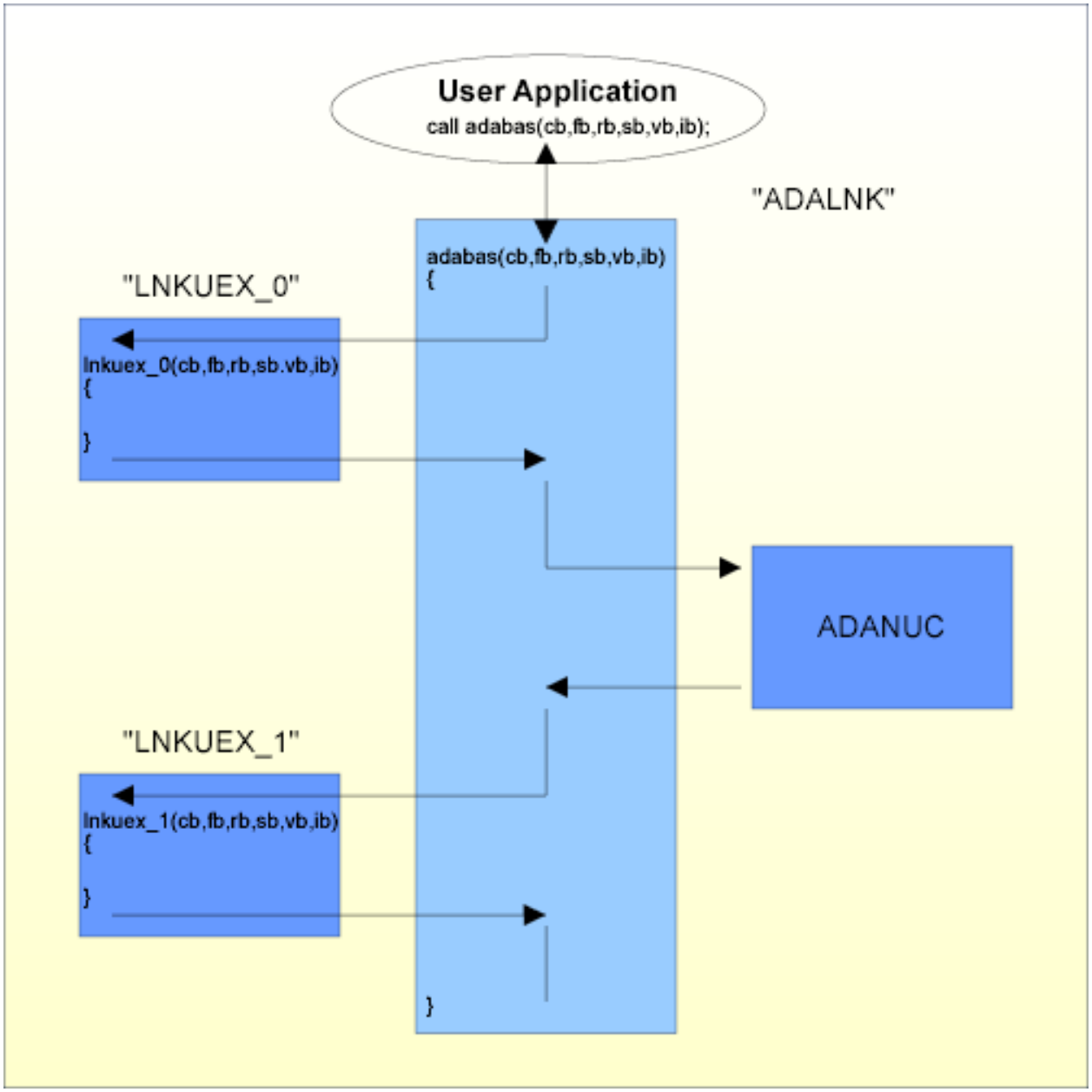
Note: The default entry function name is 'lnkuex_1(...)'.

It is also possible to define a different entry function name for each ADALNK-specific user exit, e.g.:

```
setenv LNKUEX_0 "lnk_uex.sl xx_uex_0" (UNIX)
setenv LNKUEX_1 "lnk_uex.sl yy_uex_1"
```

or

```
set LNKUEX_0=lnk_uex.dll xx_uex_0 (Windows)
set LNKUEX_1=lnk_uex.dll yy_uex_1
```



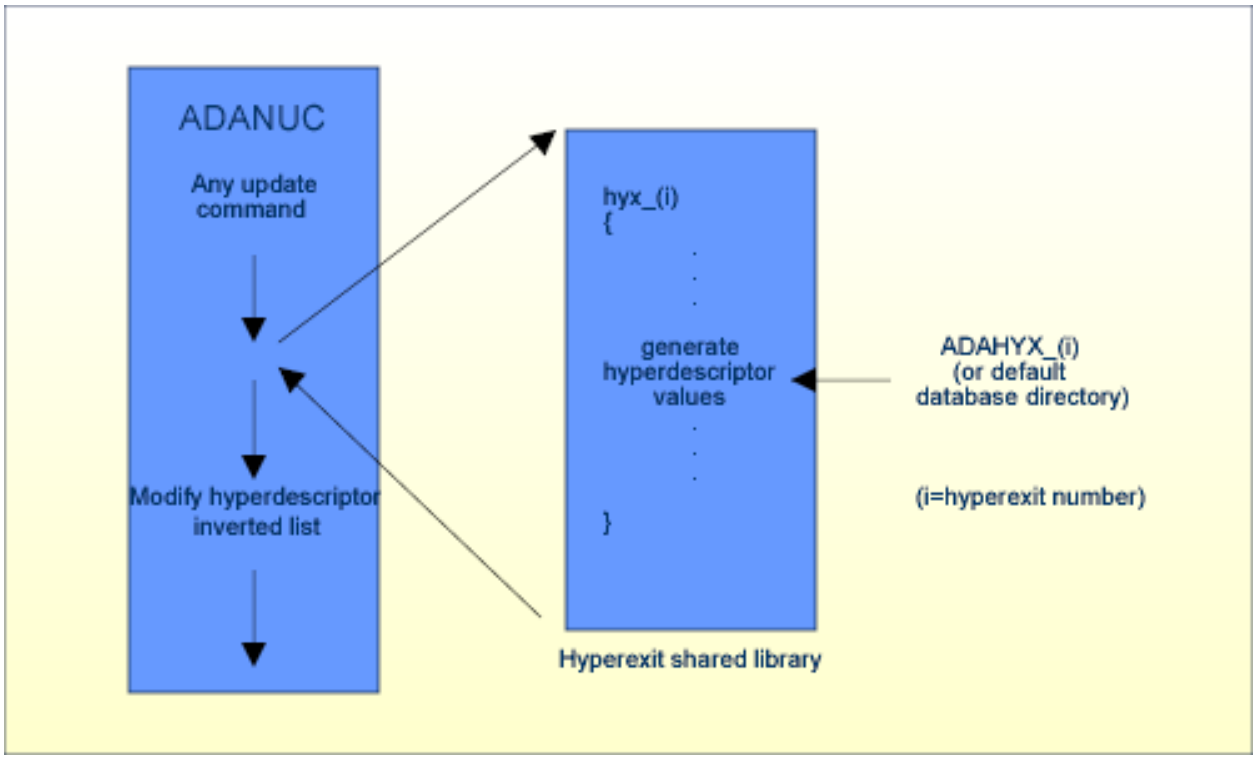
Hyperexits Overview

Like subdescriptors and superdescriptors, hyperdescriptors are descriptors that are created on the basis of parent fields in a file's FDT, but in the case of hyperdescriptors, the descriptor values are generated using a user-defined algorithm. In search criteria, you specify the generated hyper-descriptor values, except if a hyperdescriptor is defined with the HE option - in this case you specify the parent field value.

Hyperexits can modify the ISN for a hyperdescriptor value. It is also possible to specify ISNs belonging to a different file. Then you can perform a search on the file containing the hyperdescriptor without using a format buffer. In order to read the resulting records, you must change the file number to the file number of the file that contains the ISNs associated with the hyperdescriptor. If you use this feature, it is important that you specify the USERISN parameter for ADACMP and ADAMUP if you load such a file - otherwise the ISNs generated by the hyperexit will no longer fit.

This algorithm is implemented in a user-defined shared library or DLL called hyperexit and should be written in the C programming language. The basic output of a hyperexit function is one or more hyperdescriptor values.

The hyperexit is dynamically loaded at the startup of a utility or when it is accessed for the first time in the nucleus, and it is called each time that a hyperdescriptor value is generated, deleted or updated, or if a hyperdescriptor with HE option is used in a search command.



A hyperexit is called for the following actions:

- Initialize a hyperexit before it is used for the first time by the nucleus for one of the actions listed below, or during a utility run if a hyperdescriptor is found in a file's FDT.
- Insert record command: the hyperexit is called as an “after image call” to produce a hyperdescriptor value that is inserted into the inverted list. The appropriate exit function is called for all of the hyperdescriptors that are defined in the FDT.

Note: The exit function is not called if all of the hyperdescriptor's parent fields are defined with the NU/NC options and if there are no values present for the fields.

- Delete record command: the hyperexit is called as a “before image call” to produce a hyperdescriptor value that is to be deleted from the inverted list. The appropriate exit function is called for all of the hyperdescriptors that are defined in the FDT.
- Update record command: if one or more of a hyperdescriptor's parent fields is modified, the associated hyperexit is called twice (once as a “before image call” and once as an “after image call”) in order to change the hyperdescriptor value in the inverted list.
- Generate a hyperdescriptor value for a search command (only for hyperdescriptors with one parent field and defined with the HE option).

A hyperexit is used by the following utilities:

- ADAINV, for the following operations:
  - inverting a new hyperdescriptor
  - reinverting/verifying a hyperdescriptor
- ADACMP: generating hyperdescriptor values if the FDT contains a hyperdescriptor specification
- ADAULD: generating hyperdescriptor values if the FDT contains a hyperdescriptor specification.

It is important that the hyperexits generate the same values for the “after image” calls as for the “before image” calls - otherwise Adabas would not be able to remove hyperdescriptor values. This means, in particular, that following a change of a hyperexit that results in different hyperdescriptor values, the affected hyperdescriptors must be reinverted.

The hyperexits can be driven with a reentrant interface or with a non-reentrant interface. For the non-reentrant interface, the hyperexit buffers for the parent field values and the generated hyperdescriptor values are placed within the user's hyperexit module. In this case, calls to hyperexits are serialized.

For the reentrant interface, the buffers are allocated within the nucleus or utility. Therefore, a reentrant hyperexit can be called in parallel if more than one command tries to access the same hyperexit; this can improve performance compared to the non-reentrant version. In the implementation of a reentrant hyperexit, variables of storage class static may be updated only during the initialization.

Tip: The Adabas kit contains example C files for a hyperexit. These are located in the subdirectory "Adabas/examples/server" of the installation directory.

For further details about how to specify a hyperdescriptor in an FDT, see *FDT Record Structure, Hyperdescriptor*.

Hyperexit Control Block and Buffers

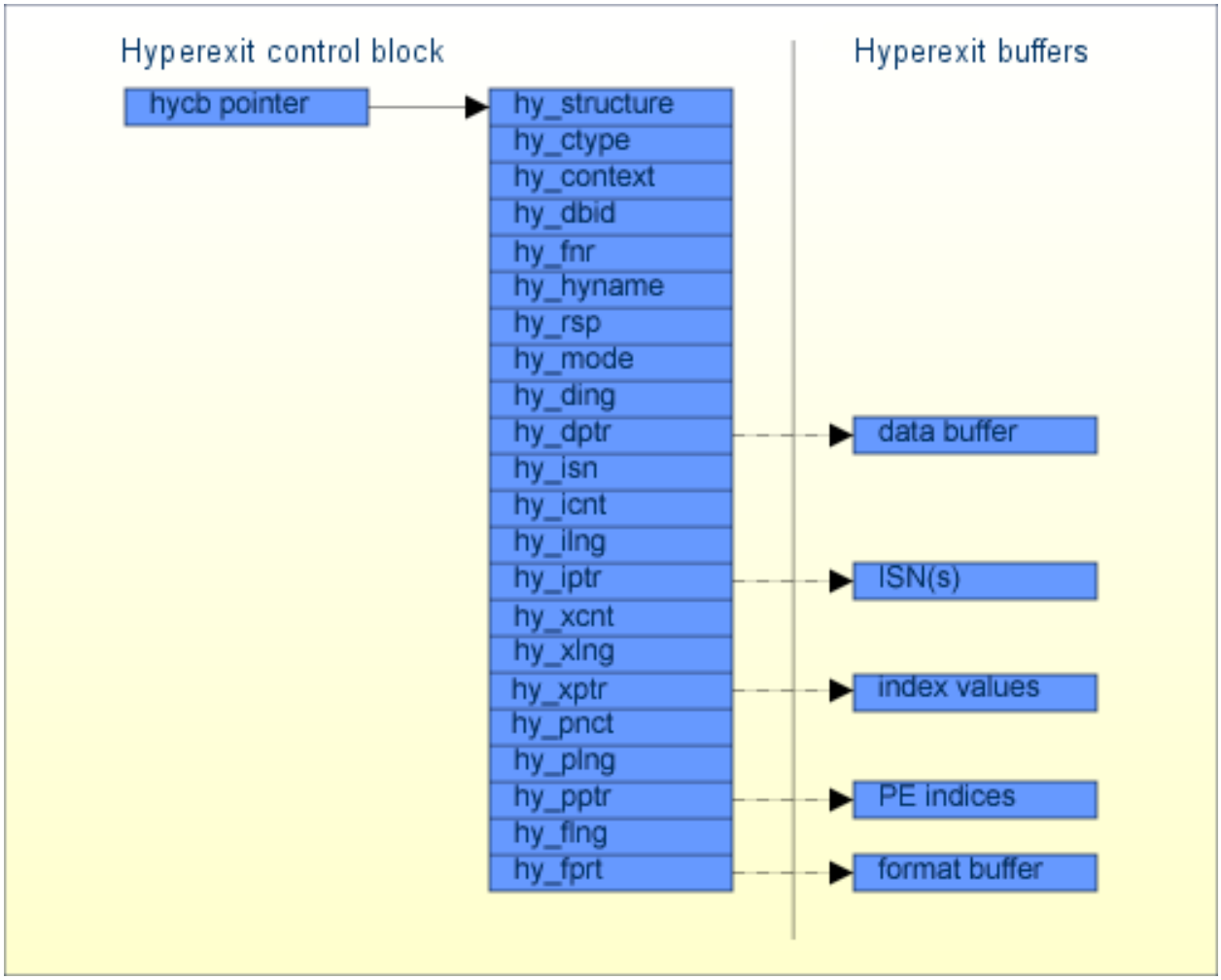
The synopsis for a hyperexit function is as follows:

```
#include <adahyx.h>

void hyx_<number> ( HYCB_ENTRY* hy_cb )
```

hy_cb is a pointer to the hyperexit control block; its C definition can be found in the header file adahyx.h in the subdirectory "Adabas/inc" of the installation directory.

The hyperexit control block is a structure that contains all of the information necessary for the hyperexit and the nucleus to handle hyperdescriptors. The structure is shown in the following diagram:



Hyperexit Control Block

Variable	Type	Description
hy_structure	unsigned char	This defines a structure level for the hyperdescriptor interface. The structure may change in future, so Adabas can check if a supported version of the hyperdescriptor interface is in use. The current structure level is 2. No other structure levels are currently supported.
hy_ctype	unsigned short	This defines which type of call is performed: 0 value generation call 1 value generation call repeated 2 initialization call 3 initialization call repeated During utility initialization or when a hyperdescriptor is processed the first time in the nucleus, an initialization call is performed. If the hyperexit needs detailed information on the parent field and the hyperdescriptor definitions, it can set the hyperexit response code hy_rsp to 1, Adabas performs an initialization call repeated. If Adabas has to convert parent values to a hyperdescriptor value, it performs a value generation call. If more than one hyperdescriptor value is to be generated, the hyperexit can deliver the values in more than one call. If additional values are to be generated, the hyperexit must set hy_rsp to 1. Then Adabas performs a value-generation call repeated.
hy_context	unsigned short	This defines the context of the value-generation call: 0 before-image value 1 after-image values 2 search-buffer value Before-image values are used when descriptor values must be removed from the index. After-image values are used when descriptor values must be inserted in the index. Search-buffer values are used if a hyperdescriptor with the HE option is specified in a search criterion of a search operation.
hy_dbid	unsigned short	This defines the database that called the hyperexit.
hy_fnr	unsigned short	This defines the hyperdescriptor's file number.
hy_hyname	char[2]	This defines the name of the hyperdescriptor.
hy_rsp	unsigned short	The response enables the hyperdescriptors hyperexit to return a success code or an error response code. The error will be converted to an Adabas nucleus response code 86 and the hyperexits value will be returned in the Additions 2 field in the control block. During an initialization call, the utility or nucleus will terminate with an error.

Variable	Type	Description
		The response codes and their meanings are as follows: **0** Success **1** Repeat call required **2-255** Reserved for Adabas **>=256** User errors
hy_mode	unsigned short	This is set by the initialization call and defines whether the exit routine is re-entrant or non-reentrant (for further information see *Reentrant / Non-reentrant Interface*) 0 (HY_NREENTRANT): non-reentrant 1 (HY_REENTRANT): reentrant
hy_dptr	unsigned char *	Data buffer pointer.
hy_dlng	unsigned int	Data buffer length. The maximum supported length is 65535.
hy_isn		This is the ISN of the data record. The ISN for the hyperdescriptor value(s) can be changed. Adabas does not check whether the ISN is less than or equal to the value of the MAXISN specified for the file.
hy_iptr	unsigned int *	ISN buffer pointer.
hy_ilng	unsigned int	The ISN buffer length. The maximum supported length is 65535.
hy_icnt	unsigned int	The number of ISN values. This number of ISNs must fit into the ISN buffer: 4 * hy_icnt must be less than or equal to hy_ilng.
hy_xptr	unsigned char *	Index value buffer pointer.
hy_xcnt	unsigned int	Number of returned index values.
hy_xlng	unsigned int	Index value buffer length.
hy_pcnt	unsigned int	Number of PE indices returned in the PE index buffer. This number must be equal to hy_xcnt if the hyperdescriptor is defined with option PE. However, in a search buffer value call, the PE index buffer is ignored.
hy_plng	unsigned int	PE index buffer length.
hy_pptr	unsigned char *	PE index buffer pointer.
hy_fprt	unsigned char *	Format buffer pointer. During an initialization call, the hyperexit may supply a format buffer. This format buffer will be used to decompress the data values of the parent field into the buffer before the hyperexit is called. The user can use the format buffer to override the default format buffer.
hy_flng	unsigned int	Format buffer length.

Data Buffer for an Initialization Call Repeated

In an Initialization Call Repeated, Adabas provides information on the parent fields and the hyperdescriptor in the data buffer. This information has the same layout as the record buffer for an LF command with command option S (for further information see *Command Reference*, *LF command*).

Bytes	Usage
1 - 2	Total length of information
3 - 4	Number of entries
5 - n	One 'F' element for each parent field
(n + 1) - m	'H' element for the hyperdescriptor

Data Buffer for a Value Generation Call

In the data buffer for a Value Generation Call/Value Generation Call Repeated, Adabas inserts the hyperdescriptor according to the format buffer specification in the Initialization Call / Initialization Call Repeated.

ISN Buffer

In a Value Generation Call / Value Generation Call Repeated, there are two ways how to define the ISN associated to the hyperdescriptor values:

1. The same ISN is to be used for all hyperdescriptor values generated by this call. In this case, the ISN must be specified in hy_isn and hy_icnt must be 0.
2. Different ISNs are used for the hyperdescriptor values generated by this call. In this case, for each descriptor value generated by the call one ISN must be inserted in the ISN buffer; hy_icnt must be equal to hy_xcnt.

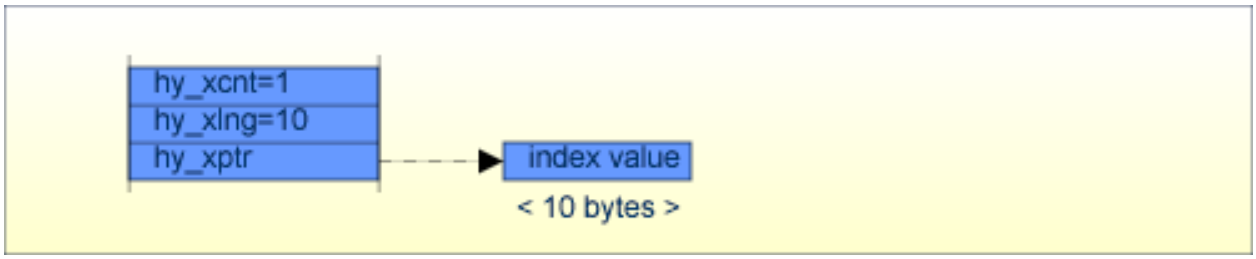
Index Value Buffer

In a Value Generation Call / Value Generation Call Repeated, the hyperexit must generate the hyperdescriptor values. hy_xcnt must be set to the number of generated hyperdescriptor values. The hyperdescriptor values must be generated in the format specified in the hyperdescriptor definition.

The following examples show how the index buffer must be generated:

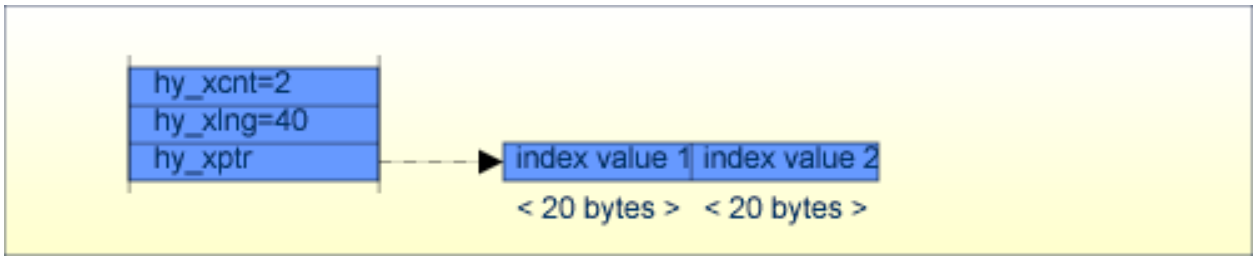
Example 1:

```
H1,10,A=HYPER(1,AA)
```



Example 2:

```
H2,20,A,MU=HYPER(1,MU)
```



For generating two hyperdescriptor values, the length of the index value buffer must be at least 40 bytes; if the maximum number of hyperdescriptor values to be generated is larger, the index value buffer must be defined accordingly larger.

PE Index Buffer

If a hyperdescriptor is defined with option PE, the hyperdescriptor must generate the PE indices corresponding to the hyperdescriptor values in the PE Index Buffer in a Value Generation Call / Value Generation Call Repeated. The number of PE indices must be equal to the number of hyperdescriptor values generated. The PE indices are stored as one byte binary; PE indices > 255 are not supported.

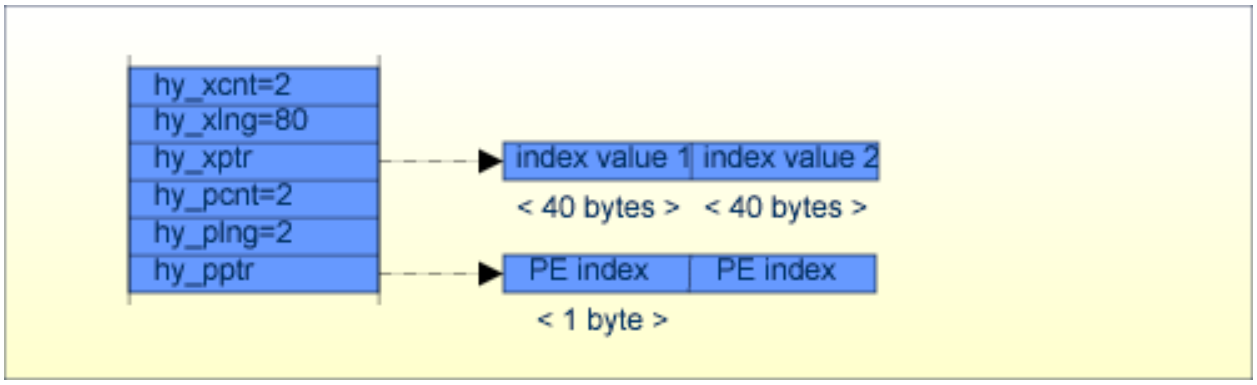
However, the hyperexit cannot supply a PE index for a search-buffer value call. The PE index from the user's search buffer will be appended in such a case.

Packed and unpacked values are checked for validity, and variable-length values are checked to ensure that they are within the maximum permitted length for their format.

The following example shows how index value buffer and PE index buffer must be generated for a hyperdescriptor defined with option PE.

Example:

```
H3,40,A,PE=HYPER(1,P1)
```



For generating two hyperdescriptor values, the length of the index value buffer must be at least 80 bytes, and the length of the PE index buffer must be at least 2; if the maximum number of hyperdescriptor values to be generated is larger, the index value buffer and the PE index buffer must be defined accordingly larger.

Format Buffer

In an Initialization Call / Initialization Call Repeated, you may supply a format buffer to be used for the decompression of the parent field values before the value generation calls of the hyperexit. The syntax for the format buffer is the same as described in Command Reference, Calling Adabas, Format and Record Buffers, Format Buffer Syntax, but there are the following restrictions for allowed field definitions:

- Only parent fields may be specified.
- If a parent field belongs to a periodic group, you may specify the periodic group count for this periodic group.

If no format buffer has been specified, Adabas assumes the following format buffer where the following format buffer elements are contained for each parent field:

- Non MU parent field not belonging to a periodic group: name
- MU parent field not belonging to a periodic group: nameC,name1-N
- Non MU parent field PA belonging to a periodic group PG: PGC,PA1-N
- MU parent field PA belonging to a periodic group PG: PGC,PA1C,PA1(1-N),PA2C,PA2(1-N),…

The following examples illustrate the use of the default format-buffer elements. The FDT shown below is used in these examples.

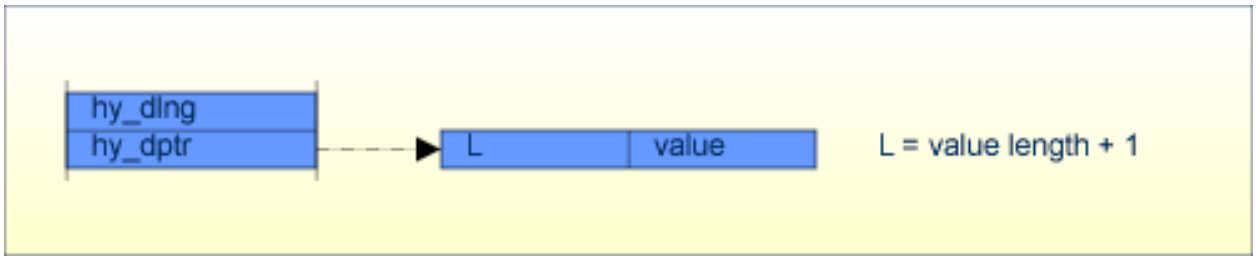
```
01,AA,0,A
01,MU,10,B,MU
01,PG,PE
 02,PP,40,A
 02,PM,20,B,MU
      .
      .
      .
```

Example 1:

```
H1,10,A=HYPER(1,AA)
```

Because the parent field AA is not an MU field and not within a periodic group, the default format buffer is:

```
AA.
```



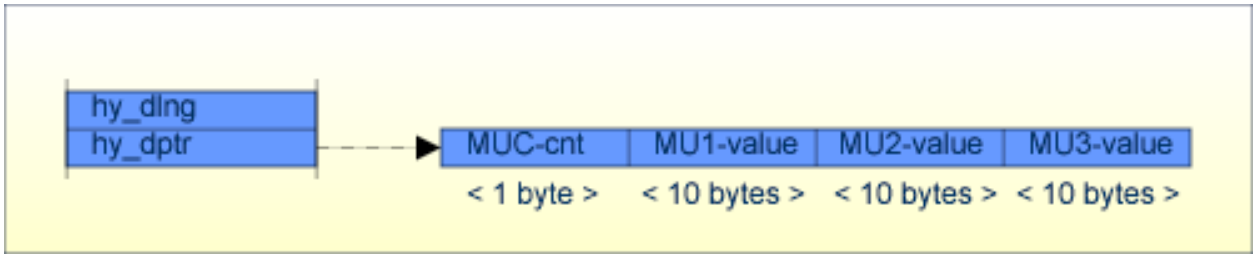
Required length = maximum expected value length + 1.

Example 2:

```
H2,20,A,MU=HYPER(1,MU)
```

Because MU is an MU field not within a periodic group, the default format buffer is:

```
MUC,MU1-N.
```



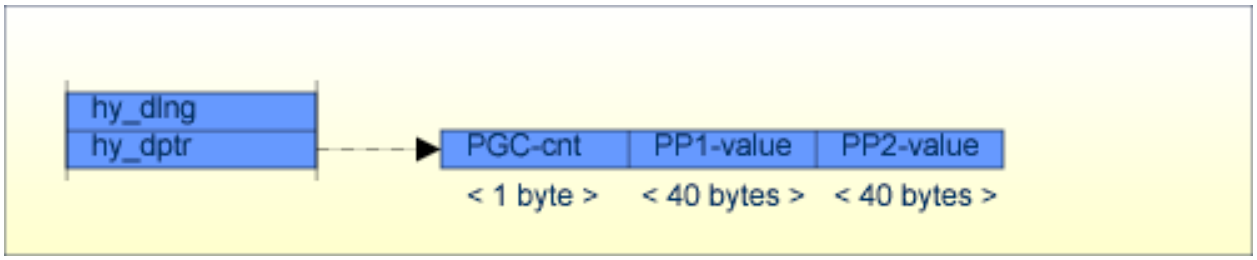
The required length is dependent of the maximum expected MU count; for this example with MU count 3 the number of required bytes is 31.

Example 3:

```
H3,40,A,PE=HYPER(1,PP)
```

Because PP is a non MU field in a PE, the default format buffer is:

```
PGC,PP1-N.
```



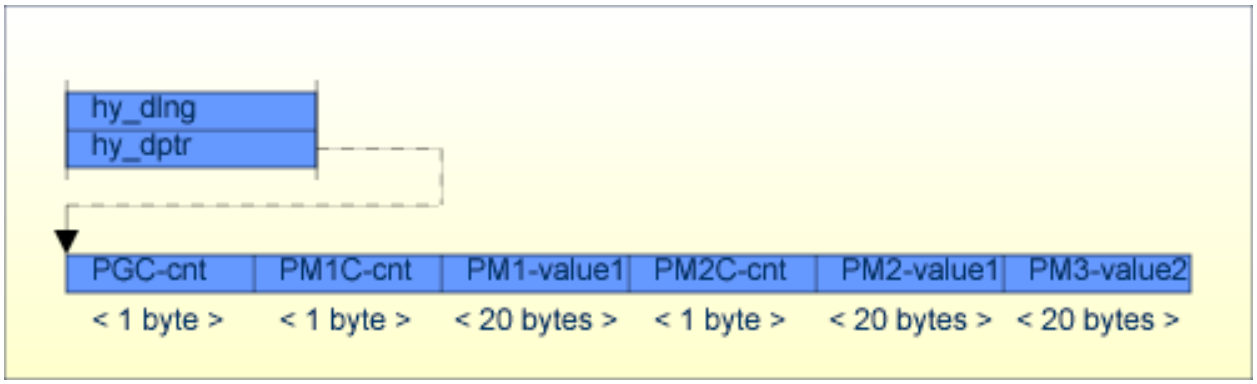
The required length is dependent of the maximum expected periodic group count; for this example with periodic group count 2 the number of required bytes is 81.

Example 4:

```
H4,40,A,PE,MU=HYPER(1,PM)
```

Because the parent field is an MU field in a periodic group, the default format buffer depends on the periodic group count in this example; if the periodic group count is 2, the format buffer is:

```
PGC,PM1C,PM1(1-N),PM2C,PM2(1-N).
```



The required length for the data buffer depends on the maximum expected periodic group count and the expected MU counts; the required length for the example data with PGC=2, PM1C=1, PM2C=2 is 63.

Hyperexit Interfaces

Reentrant/Non-Reentrant Interfaces

The hyperexits can be implemented with a reentrant or a non-reentrant interface.

A computer routine is called reentrant if it can be safely executed concurrently; that is, the routine can be reentered while it is already running in another thread. In the implementation of a reentrant hyperexit, variables of storage class static may be updated only during the initialization, because otherwise one thread could overwrite the data created by another thread. This also means that the calling nucleus or utility must provide the hyperexit buffers - each thread calling the hyperexit must provide its own hyperexit buffers. An exception is the format buffer; it is always provided by the hyperexit - also for the reentrant interface, because it is used only during the initialization.

For the non-reentrant interface, Adabas serializes the hyperexit calls. This serialization of hyperexit calls can decrease the performance. The hyperexit buffers used must be provided by the hyperexit in this case.

The different handling of the hyperexit buffers has the consequence that the different calls must be implemented differently for reentrant or non-reentrant hyperexits.

In the initialization call, the hyperexit states whether it is running in reentrant or non-reentrant mode. For reentrant mode, it must set the field "hy_mode" to HY_REENTRANT (for the reentrant version) . For non-reentrant mode hy_mode must be equal to HY_N_REENTRANT - this is the default.

Hyperexit Calls

Fields in the hyperexit control block not mentioned as output parameter in the following should not be modified during a hyperexit call.

Initialization Call

Input Parameters (reentrant and non-reentrant)

hy_ctype	This value is 2 for an initialization call.
hy_dbid	The number of the database.
hy_fnr	The number of the hyperdescriptor.
hy_hyname	The name of the hyperdescriptor.

Output Parameters (non-reentrant)

hy_structure	The actual structure number is 2.
hy_rsp	Success code or an error response code. In particular, you can use at the init call hy_rsp=1 to initiate an “Initialization Call Repeated”. If hy_rsp > 1during an initialization call, the utility or nucleus will terminate with an error. (Response 79 - Hyperexit not available). The Nucleus Log gets the error-message: %ADANUC-W-HYERR, HYPER userexit, descriptor HY, file ..., reason=....
hy_mode	HY_N_REENTRANT
hy_dlng	Data buffer length. The data buffer must be defined large enough for the parent fields.
hy_dptr	Data buffer pointer for parent field values.
hy_flng	Own format buffer length or 0, if no format buffer provided .
hy_fptr	Own format buffer pointer or NULL, if no format buffer provided.
Format buffer	Own format buffer, optional.

Output Parameters (reentrant)

hy_structure	The actual structure number is 2.
hy_rsp	Success code or an error response code. In particular, you can use at the init call hy_rsp=1 to initiate an “Initialization Call Repeated”. If hy_rsp > 1during an initialization call, the utility or nucleus will terminate with an error. (Response 79 - Hyperexit not available). The Nucleus Log gets the error-message: %ADANUC-W-HYERR, HYPER userexit, descriptor HY, file ..., reason=....
hy_mode	HY_REENTRANT
hy_dlng	Data transfer buffer length. At the init call you have to set the data buffer length.
hy_xlng	Index value buffer length
hy_ilng	ISN value buffer length (0 = not used).
hy_plng	Periodic group index buffer length (0 = not used).
hy_flng	Own format buffer length or 0, if no format buffer provided .
hy_fptr	Own format buffer pointer or NULL, if no format buffer provided.
Format buffer	Own format buffer, optional.

Initialization Call (repeated)

If_hy_rsp was set to 1 in the initialization call, Adabas calls the hyperexit call again with hy_ctype=3. In the data buffer, Adabas returns information on the hyperdescriptor and each parent field as described above.

Value generation Call

Input Parameters (non-reentrant)

hy_ctype	0 indicates Value Generation Call.
hy_dbid	The database ID.
hy_fnr	The file number of the file containing the hyperdescriptor.
hy_hyname	The name of the hyperdescriptor.
hy_dlng	Data buffer length.
hy_dptr	Pointer to data buffer.
hy_isn	ISN of record.
Data buffer	Parent field values.

Output Parameters (non-reentrant)

hy_rsp	Success code or an error response code. In particular, you can use at the value generation call hy_rsp=1 to initiate a “Value generation call repeated”. If hy_rsp > 1 the hyperexit returns an error response code. The error will be converted to an Adabas nucleus response code 86 and the hyperexit's value will be returned in the Additions 2 field in the control block.
hy_cnt	Number of index values.
hy_xlng	Length of index value buffer.
hy_xptr	Pointer to index buffer.
hy_icnt	Number of ISN values (0 = not used).
hy_ilng	ISN buffer length (0 = not used).
hy_iptr	Pointer to ISN buffer.
hy_pcnt	Number of PE indices (0 =not used).
hy_plng	Length of PE index buffer (0 = not used)
hy_pptr	Pointer to PE index buffer (NULL = not used).
Index value buffer	Hyperdescriptor values as described above.
ISN buffer	ISNs associated with each descriptor value, optional.

PE index buffer	PE indices; only if hyperdescriptor defined with option PE and no search buffer value call.

Input Parameters (reentrant)

hy_structure	The actual structure number is 2.
hy_ctype	After the value generation call it is 0.
hy_dbid	The number of the database.
hy_fnr	The file number of the file containing the hyperdescriptor.
hy_hyname	The name of the hyperdescriptor.
hy_dptr	Pointer to parent field values.
hy_dlng	Data transfer buffer length.
hy_isn	ISN of record.
hy_ilng	Length of ISN buffer.
hy_iptr	Pointer to ISN buffer.
hy_xlng	Length of index values buffer.
hy_xptr	Pointer to index values buffer.
hy_plng	Length of periodic counts buffer.
hy_pptr	Pointer to periodic counts buffer.

Output Parameters (reentrant)

hy_rsp	Success code or an error response code. In particular, you can use at the value generation call hy_rsp=1 to initiate a “Value generation call repeated”. If hy_rsp > 1 the hyperexit returns an error response code. The error will be converted to an Adabas nucleus response code 86 and the hyperexit's value will be returned in the Additions 2 field in the control block.
hy_icnt	Number of ISN values.
hy_xcnt	Number of index values.
hy_pcnt	Number of periodic count values.
Index value buffer	Hyperdescriptor values as described above.
ISN buffer	ISNs associated with each descriptor value, optional.
PE index buffer	ISNs associated with each descriptor value, optional.

Value Generation Call Repeated

If hy_rsp is set to 1 during a value generation call, Adabas will process the index values and then recall the hyperexit with the same data-buffer contents, (reentrant and non-reentrant), with hy_ctype=1. Then the hyperexit is able to generate more hyperdescriptors values, or to generate hyperdescriptors values for a different ISN.

Creating and Defining User Exits and Hyperexits

The nucleus and the utilities activate a user exit by using the parameter USEREXIT (in the utilities ADACMP and/or ADADCU). If this parameter is set, the user exit is called at specific points in the processing.

Hyperexits are activated if a file in which hyperdescriptors are defined is updated.

A user exit or hyperexit is defined by performing the following steps:

1. Write the user exit or hyperexit in the C programming language. The header files adauex.h (for user exits) and adahyx.h (for hyperexits) should be used.

   The exits can be written with default function names. The convention is "uex_"/"hyx_" followed by the number of the user exit or hyperexit, e.g.

   ```
   uex_1()        /* Default name user exit 1 */
   {}

   hyx_4()        /* Default name hyperexit 4 */
   {}
   ```

   Other names can be used by setting environment variables/logical names (see 4 below).
2. Compile the source file of the user exit or hyperexit; the option for position-independent code must be used.
3. Link the user exit or hyperexit as a shared library. The linker options used to create a shared library are machine-dependent.
4. Make the user exit or hyperexit available to Adabas by connecting the shared library with an environment variable/logical name. You may specify either the member name of the shared library, if the directory containing the shared library is contained in the LD_LIBRARY_PATH (UNIX) or PATH (Windows), or you may specify the absolute path of the shared library. The environment variable is “ADAUEX_” (for the user exit) or “ADAHYX_” (for the hyperexit) followed by the number of the user exit or hyperexit, e.g.

   In csh:

```
setenv ADAUEX_1 userex1.sl  # connect user exit 1 (see note below)
setenv ADAHYX_4 $ADADATADIR/exits/hypex4.sl   # connect hyperexit 4 (see note ↩
below)
```

Note: Depending on the UNIX derivative used, the shared library extension is "sl" or "so".

In sh or ksh:

```
ADAUEX_1=userex1.sl
export ADAUEX_1
```

For Windows:

```
set ADAUEX_1=userex1.dll  # connect user exit 1
set ADAHYX_4=%ADADATADIR%\exits\hypex4.dll   # connect hyperexit 4
```

For the sake of convenience, the default function names for the user exit/hyperexit can be overwritten, e.g.

```
my_uex_1 ()
{}

nga_hyx_4 ()
{}

setenv ADAUEX_1 "userex1.sl my_uex_1" (UNIX platforms)
setenv ADAHYX_4 "$ADADATADIR/exits/hypex4.sl nga_hyx_4"

set ADAUEX_1=userex1.dll;my_uex_1 (Windows)
set ADAHYX_4=%ADADATADIR%\exits\hypex4.dll;nga_hyx_4
```

If the nucleus cannot find the exit as specified in the environment variables, it continues searching as described in the following step.

5. The use of the ADAHYX environment variable can be omitted if the hyperexit shared libraries are located in the default database directory, with the following naming convention:

   UNIX platforms:

   ```
   $ADADATADIR/db<xxx>/adahyx_<i>.<ext>
   ```

   where xxx is the database id, i is the hyperexit number, and ext is the shared library extension ("sl" or "so").

Windows:

```
%ADADATADIR%\db<xxx>\adahyx_<i>.dll
```

where xxx is the database id, i is the hyperexit number.

Example (UNIX):

```
$ADADATADIR/db076/adahyx_1.sl
```

Example (Windows):

```
%ADADATADIR%\db076\adahyx_1.dll
```

In order to load a shared library, Adabas first takes the corresponding ADAHYX environment variable. If this is not present, Adabas then searches in the default database directory ($ADADATADIR/db<xxx> or %ADADATADIR%\db<xxx>).

The Adabas kit contains the required C header files, example sources for user exits and hyperexits and a corresponding makefile. Because the ADALNK user exits may be used on a different platform than the server platform, the ADALNK user exit, the corresponding C header file and makefile are also provided with Entire Net-Work (see the Entire Net-Work documentation for more information).

UNIX Platforms

The required C header files are located located in the directories $ADAPROGDIR/inc and $ACLDIR/$ACLVERS/inc - nucleus and utility user exits need the include file adauex.h, ADALNK user exits need lnkuex.h and hyperexits need adahyx.h.

Example source files for nucleus user exit 1, nucleus user exit 2, nucleus user exit 4, ADACMP user exit 6, ADAULD user exit 7 and hyperexit 1 and the corresponding makefile are located in $ADAPROGDIR/examples.

Enter the following in order to build one of these user exit examples:

```
cd  $ADAPROGDIR/examples

make target
```

where *target* is one of the following:

User exit/hyperexit	Example source file name	*target*
Nucleus user exit 1	adauex1.c	uex1
Nucleus user exit 2	adauex2.c	uex2
Nucleus user exit 4	adauex4.c	uex4
ADACMP user exit 6	adauex6.c	uex6
ADAULD user exit 7	adauex7.c	uex7
Hyperexit 1	adahyx1.c	hyx1

The shared library for the user exit or hyperexit is created in $ADAPROGDIR/examples/server.

An example source file for ADALNK user exit 0 and ADALNK user exit 1 (file name: lnkuex.c) and the corresponding makefile are located in $ACLDIR/examples.

Enter the following in order to build this user exit example:

```
cd  $ACLDIR/examples/client

make lnkuex ↩
```

The shared library for the user exit is created in $ACLDIR/examples.

Windows Platforms

The required C header files are located in the directories %ADAPROGDIR%\inc and %ACLDIR%\inc - nucleus and utility user exits need the include file adauex.h, ADALNK user exits need lnkuex.h and hyperexits need adahyx.h.

Example source files for nucleus user exit 1, nucleus user exit 2, nucleus user exit 4, ADACMP user exit 6, ADAULD user exit 7 and hyperexit 1 and the corresponding makefile are located in %ADAPROGDIR%\examples.

Enter the following in order to build one of these user exit examples:

```
cd  %ADAPROGDIR%\examples

nmake target
```

where *target* is one of the following:

User exit/hyperexit	*target*
Nucleus user exit 1	uex1
Nucleus user exit 2	uex2
Nucleus user exit 4	uex4
ADACMP user exit 6	uex6
ADAULD user exit 7	uex7
Hyperexit 1	hyx1

An example source file for ADALNK user exit 0 and ADALNK user exit 1 (file name: lnkuex.c) and the corresponding makefile are located in %ACLDIR%\examples.

Enter the following in order to build this user exit example:

```
cd  %ACLDIR%\examples

nmake lnkuex ↩
```

The DLL for the user exit or hyperexit is created in %ACLDIR%\examples.

New Hyperexits while Nucleus is Active

A new hyperexit can be activated via a utility with the nucleus already active. This can happen

- when a new file is defined (ADAFDU),
- when a file is restored or overlaid (ADABCK),
- when a file is imported (ADAORD) or
- when a hyperdescriptor is created for an existing file with ADAINV

The nucleus initializes the hyperexit before the first value-generation call.

In the case of ADAFDU, if defining a new file and when the FDT contains a hyperdescriptor, it is possible to transfer a valid ADAHYX environment variable to the nucleus:

Example

```
setenv ADAHYX_1 /user/adabas/hyper1.sl
setenv FDUFDT file20.fdt
adafdu < file20.fdu
```

If the ADAHYX environment variable is not specified or if a file is restored using ADABCK, the nucleus will search for the hyperexit in the location specified at nucleus startup (see above). Hence, the corresponding shared library or DLL must be moved to the appropriate directory.

10 Adabas On Read-only Devices

Adabas databases can reside on read-only devices. If at least ASSO1 is located on a read-only device, the whole database is treated as read-only.

Restrictions when using the Adabas Nucleus

For read-only databases, there are several restrictions concerning the use of the Adabas nucleus.

A nucleus runs in read-only mode if ASSO1 is located on a read-only device, or if ASSO1 cannot be opened for writing, or if the nucleus session is started with the Adabas nucleus parameter OPTIONS=READONLY. In the first case, the nucleus switches automatically to read-only mode, regardless of the setting of the OPTIONS parameter. The second case can be used to run a nucleus session in read-only mode if the database is not located on a read-only device.

A pending autorestart check is always done when a writeable WORK container is specified. If an autorestart is pending, a read-only database session cannot be started.

If the nucleus is started in read-only mode, the database ID that is given as a startup parameter is not checked against the ID of the real database. Instead, the environment variable settings for the database container files identify the database. This means that a read-only database can run with any valid database ID. Therefore, when starting the nucleus or a utility, any number can be assigned to the database ID parameter. Applications that communicate with the nucleus must use this number to access the database.

Databases which are partially read-only and for which ASSO1 is writeable are not supported by Adabas.

READONLY option

If the nucleus parameter OPTION=READONLY is set, the nucleus runs in read-only mode. The database can be located on a read-only or on a writeable device. If ASSO1 is on a read-only device, ADANUC will automatically switch to read-only mode, even if the parameter OPTION=READONLY is not set. In this case, a warning message will be displayed.

The read-only mode causes

- update commands (A1, E1, N1, N2) to be disabled
- checkpoint writing to be disabled (C1 and utility / EXU checkpoints)
- logging to be disabled (start of nucleus with NOPLOG is required)
- the hold logic to become inactive (The commands L4, L5, L6, S4 are accepted, but have no effect on the hold queue, i.e. they work exactly like L1, L2, L3, S1, respectively. The commands BT, ET, HI, RI are accepted but have no effect).
- ET/CL commands with user data to be forbidden

All file open modes that can be specified in the OP command are allowed and therefore all permitted user types are also allowed. Note, however, that exclusive control users do not write OPEN and CLSE checkpoints.

The session number is not increased by a nucleus session running in read-only mode. Command logs are still supported but the command log number is not increased.

For the WORK container there are two options:

1. Although not required, there is a WORK container.
2. There is no WORK. In this case, the environment variable WORK1 must be set to "READONLY".

Restrictions when using Adabas Utilities

A utility can process a read-only database if a nucleus with the specified database ID is running in read-only mode or if the ASSO1 container file is located on a read-only device. In neither case is the database ID that is given as a parameter checked against the real database ID (same as for nucleus processing described above). However, to enable communication between a running nucleus and utility, they must be started with the same database ID (which may differ from the stored ID of the database).

Note that if the database IDs specified for the nucleus and the utility are different, the communication link fails and the utility runs in offline mode on the same database as the nucleus.

A pending autorestart check is always done when a writeable WORK container is specified. If the WORK1 environment variable is set to READONLY, the ASSO1 container must be located on a read-only device or the nucleus must be running with OPTIONS=READONLY, otherwise the utility will be terminated with the error message RDONP ("Dataset WORK1, READONLY is not permitted").

Some utilities cannot be used with a read-only database, since their purpose is to modify the database. The open database calls for these utilities are terminated with the error message UNPRRD (“Readonly database, utility not permitted to run”). The utilities for which this applies are:

- ADACVT
- ADADBM
- ADAFDU
- ADAREC

For other utilities, the function set is restricted. These utilities display the warning RONLYDB (“Readonly database, some functions disabled”).

The following utilities are affected:

- ADABCK: the RESTORE and OVERLAY functions cannot be used.
- ADAINV: only the SUMMARY and VERIFY functions can be used.
- ADAMUP: only the SUMMARY function can be used.
- ADAORD: only the EXPORT function can be used.
- ADAOPR:
  - the LOCK and UNLOCK functions cannot be used.
  - FEOF=CLOG is possible, but the increased CLOG number will not be propagated to the next nucleus session, i.e., all subsequent read-only nucleus sessions will start with the same CLOG number.
- ADASCR: only the DISPLAY function can be used.

When the nucleus runs in read-only mode, or utilities are executed against a database for which ASSO1 resides on a read-only device, they do not check the database ID given as a parameter against that of the database. However, to enable communication between the nucleus and the utilities, they must be started with the same database ID (which may differ from that stored in the database).

Read-only raw sections are not supported.