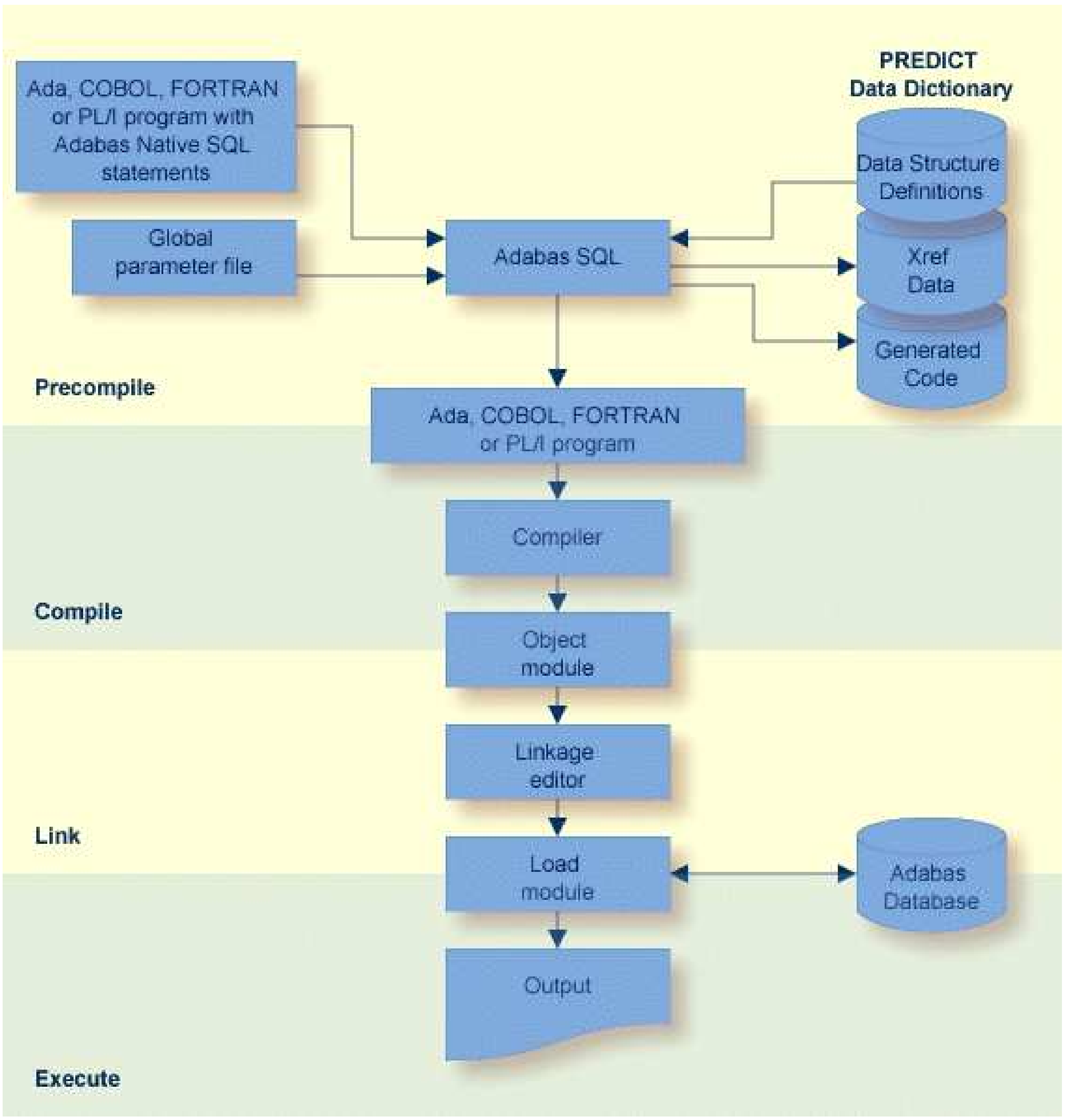
PROGRAMMING CONSIDERATIONS

Using Adabas Native SQL does not require you to learn new programming techniques. Programs are written in Ada, COBOL, FORTRAN77 or PL/I as before, with Adabas Native SQL statements that access the Adabas database inserted at the required places. The Adabas Native SQL preprocessor converts the Adabas Native SQL statements into comments, inserts the generated code and data structures into the source stream and passes the remainder of the program through without alteration. At the same time, Adabas Native SQL optionally writes to the data dictionary a cross-reference list of the files and fields used by the program.



This chapter covers the following topics:

Rules for Adabas Native SQL Statements

Each Adabas Native SQL statement is preceded by "EXEC ADABAS". Each Adabas Native SQL statement is terminated by "END-EXEC" (in Ada, COBOL or FORTRAN), or by "END-EXEC" or ";" (in PL/I). These delimiters enable the preprocessor to distinguish Adabas Native SQL statements from regular Ada, COBOL, FORTRAN or PL/I code. The following COBOL program includes two Adabas Native SQL statements:

```
IDENTIFICATION DIVISION.
PROGRAM ID. EXAMPLE.
AUTHOR. SAG.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    SKIP2
    EXEC ADABAS
         BEGIN DECLARE SECTION
    END-EXEC
PROCEDURE DIVISION.
    EXEC ADABAS
```

```
          SELECT NAME, AGE, SALARY
          FROM PERSONNEL
          WHERE NUMBER-OF-DEPENDENTS GT 4
     END-EXEC
     DISPLAY NAME AGE SALARY
     GOBACK.
```

"EXEC ADABAS" must be specified within one line. The same is true for "END-EXEC". Only one Adabas Native SQL statement may be written between "EXEC ADABAS" and "END-EXEC". The Adabas Native SQL statement is restricted to a maximum of 100 lines in length (including "EXEC ADABAS" and "END-EXEC").

Mixing Adabas Native SQL statements and regular source code statements is not allowed; Ada, COBOL, FORTRAN or PL/I code or comments should not appear between "EXEC ADABAS" and the corresponding "END-EXEC".

Note:
(for COBOL users): The generated statements may include periods to terminate internal `IF` statements. Adabas Native SQL statements are therefore not permitted within `IF...ELSE` sections. This restriction does not apply to programs generated with the global parameter `LANG COBOL/II or LANG COBOL/LE`; in this case, Adabas Native SQL generates `END-IF` statements instead of periods, so there are no restrictions on nesting Adabas Native SQL statements within other `IF...ELSE...END-IF` statements.

Note:
(for COBOL/II or COBOL/LE users): Adabas Native SQL will generate an extra statement with a period at the end while generating a SQL statement in COBOL/II in the case that the END–EXEC clause ends with a period: END–EXEC. In this case, users can “ask” Adabas Native SQL to generate a period at the end of the generation.

Source Program Maintenance

The source program stored in the programmer’s library includes Adabas Native SQL statements, but not the code they generate. Therefore every compilation must be preceded by a pass through the Adabas Native SQL preprocessor. The preprocessor produces as its output a program in Ada, COBOL, FORTRAN or PL/I, including the original Adabas Native SQL statements, which are now marked as comments. This program should now be compiled and link-edited in the normal manner. In the compiler listing, the generated statements are identified in columns 73..80 by the characters "ADABAS" (executable code and internal data) or "ADADATA" (data definitions that are of use to you). This identification enables you to locate the lines that contain the data definitions easily.

Note:
Do not alter variables that are declared in lines marked "ADABAS". You should only use those variables that are declared in lines marked "ADADATA".

Ada, FORTRAN and IBM PL/I source files may include line numbers in columns 73..80. COBOL source files may include line numbers in columns 1..6 and/or 73..80. Adabas Native SQL preserves this line-numbering, which serves as a cross-reference between the source code in the programmer’s library and the compiler listing. The line sequence numbers are also used by the response code interpretation report and the TRACE report to help you when debugging.

PL/I source files in VMS environments may not include line numbers.

If the source code is not numbered, Adabas Native SQL automatically generates line numbers in columns 73..80.

The first Adabas Native SQL statement in the program must be the following:

```
EXEC ADABAS
    BEGIN DECLARE SECTION
END-EXEC
```

Adabas Native SQL generates all the variables including the Adabas buffers after this statement.

Note for COBOL users: This statement must be in the WORKING-STORAGE SECTION of the DATA DIVISION.

The Record Buffer and Reference to Data

A record buffer is an area of storage in the user's program that is used by Adabas to transfer information to or from the database. Whenever an Adabas read command is executed, the desired database fields are located and copied into the record buffer.

Note:
No record buffer is generated for FORTRAN programs; however, there is a character string which encompasses all fields and serves the same purpose as a record buffer. Throughout this document, the term *record buffer* is used; however if a FORTRAN program is being discussed, this term should be interpreted as the character string referred to above.

Referencing Database Fields

To use data in database fields, refer to it using qualified identifiers composed of the record buffer name together with the basic field name as defined in the data dictionary. See table below.

Language	Form of Reference
Ada, PL/I	BUFFER.FIELD
COBOL	FIELD OF BUFFER
FORTRAN	No qualification possible

Note:
If more than one database field is used, a prefix or suffix (in the SELECT statement itself) should be used to make the name unique.

If the Adabas Native SQL statement that causes the record buffer to be generated does not have an alias name in the `FROM` clause, then the level-1 record buffer name is the same as the (first) file name. If the `FROM` clause does include an alias name, then the alias name is used as the level-1 record buffer name. Levels are not used in Ada or FORTRAN.

Adabas Native SQL generates a name at level 2 for internal use only. Do not use this name in your programs.

Synonyms

The field names are generated beginning at level 3. The variable names that Adabas Native SQL generates are taken from Predict. If the program is written in Ada and an Ada field name synonym is defined in the data dictionary, then the synonym is used to generate the field name in the Adabas Native SQL record buffer. If the program is written in COBOL, FORTRAN or PL/I, then the COBOL, FORTRAN or PL/I field name synonym is used respectively. If no field name synonym is defined for the language in which the program is written, the basic name of the field is used. Note that the cross-reference information written to the data dictionary by Adabas Native SQL is always the basic name of the field and not the language-dependent synonym.

Prefix/Suffix

Having selected the field name or synonym, Adabas Native SQL then attaches the prefix and suffix to the name. These are taken from one of the following sources:

Source	Description
Local (highest priority)	Use the PREFIX and SUFFIX options for the current COMPARE, FIND, HISTOGRAM, INSERT, READ, SORT or UPDATE statement.
Global	Use the PREFIX and SUFFIX clauses of the global Adabas Native SQL OPTIONS parameter (see page)
Predict (lowest priority)	The current generation defaults for the respective language are used.

The first two options can only be used if the appropriate field in the Predict Modify...Defaults screen for Ada, COBOL, FORTRAN or PL/I is marked with an "X", indicating it may be modified by the user. Otherwise the prefix and suffix values defined in the data dictionary cannot be overridden.

Validation

The field name is now validated by examining it for characters that do not conform with the rules for forming identifiers in the appropriate language (Ada, COBOL, FORTRAN or PL/I). If any illegal characters are found, they are processed according to the setting of the ’validation character’. See table below:

Validation Character	Result
Null string (two consecutive apostrophes) in global parameter or Blank (Predict default)	Invalid characters in a field name will result in an error message but will not be modified.
Replace character (letters A-Z, digits 0-9 or special character depending on language)	Invalid characters in a field name are replaced by this character.
Asterisk	Invalid characters in the field name are deleted.

The validation character is taken from one of the following sources:

Source	Description
Global (higher priority)	Use the `VALIDATION` clause of the global `OPTIONS` parameter of Adabas Native SQL. Only possible if the field Validate in the Predict Modify...Defaults screen is marked with an "X".
Predict	The current generation default for the respective language is used.

Truncation

If the field name is now too long, it is truncated by deleting characters from the left, middle or right, and a warning message is issued. The truncation character is taken from one of the following sources:

Source	Description
Global	Use the `TRUNCATION` clause of the global Adabas Native SQL `OPTIONS` parameter. Only possible if the field Truncation in the Predict Modify...Defaults screen is marked with an "X".
Predict	The current generation default for the respective language is used.

Field Attributes

The attributes of the variables (format, length, etc.) are also taken from the data dictionary. If the definition does not conform to the Ada, COBOL, FORTRAN or PL/I standards, the field is declared as an alphanumeric field. (Examples of non-conforming definitions would be 3 bytes binary or 5 bytes binary.)

Example: If there are fields called NAME and CITY in the Adabas file PERSONNEL, the following Adabas Native SQL statement-fragment is valid:

```
SELECT NAME, CITY
FROM PERSONNEL
```

You may refer to the variables in the record buffer as:

```
PERSONNEL.NAME, PERSONNEL.CITY               (Ada)
NAME OF PERSONNEL, CITY OF PERSONNEL         (COBOL)
NAME, CITY                                   (FORTRAN)
PERSONNEL.NAME, PERSONNEL.CITY               (PL/I)
```

If you use the alias name option:

```
SELECT NAME, CITY
FROM PERSONNEL PERSON-ALIAS
```

then Adabas Native SQL generates a record buffer structure with the name PERSON_ ALIAS (Ada, PL/I) or PERSON-ALIAS (COBOL). You may refer to the variables in the record buffer as:

```
PERSON_ALIAS.NAME, PERSON_ALIAS.CITY         (Ada)
NAME OF PERSON-ALIAS, CITY OF PERSON-ALIAS   (COBOL)
NAME, CITY                                   (FORTRAN)
PERSON_ALIAS.NAME, PERSON_ALIAS.CITY         (PL/I)
```

Note:
(for FORTRAN users): Qualification is not possible in FORTRAN. However, if the database field is used in more than one Adabas Native SQL statement, a prefix or suffix (in the statement itself) must be used to make the name unique.

Note:
(for Ada and FORTRAN users): Numeric fields are transformed into character fields; therefore, whenever these fields are initialized and whenever values are assigned to these fields, the values must be filled with leading zeros, for example, "0001".

Groups

If the name specified is the name of a group (GR), Adabas Native SQL automatically generates declarations for the lower-level fields at all levels, in accordance with the definition stored in the data dictionary. The field names will be the full field names as defined in the data dictionary. If Ada, COBOL, FORTRAN or PL/I synonyms are defined in the data dictionary, they will be used in place of the full field names.

Example:

```
SELECT PERSON
FROM PERSONNEL
```

The structure of the Ada record buffer is as follows:

```
type RECORD_BUFPERS is
 record

      NAME                 : STRING (1..20);
      FIRST_NAME           : STRING (1..15);
      INITIAL              : STRING (1..1);
      SEX                  : STRING (1..1);
      AGE                  : STRING (1..2);
      FAMILY_STATUS        : STRING (1..10);
      NUMBER_OF_DEPENDENTS : STRING (1..2);
      ISN                  : INTEGER;
      QUANTITY             : INTEGER;
      RESPONSE_CODE        : SHORT_INTEGER;

end record;
PERSONNEL: RECORD_BUFPERS;
```

The structure of the COBOL record buffer is as follows:

```
01  PERSONNEL.
    02 RECORD-BUF-0-1.
     03 PERSON.
      04 NAME                        PIC X(20).
      04 FIRST-NAME                  PIC X(15).
      04 INITIAL                     PIC X(1).
      04 P-DES.
       05 SEX                        PIC X(1).
       05 AGE                        PIC 9(2).
       05 FAMILY-STATUS              PIC X(10).
       05 NUMBER-OF-DEPENDENTS       PIC 9(2).
    02 ISN                           PIC 9(9) COMP.
    02 QUANTITY                      PIC 9(9) COMP.
    02 RESPONSE-CODE                 PIC 9(4) COMP.
```

The FORTRAN equivalent is as follows:

```
CHARACTER*    20 NAME
CHARACTER*    15 FNAME
CHARACTER*     1 INIIAL
CHARACTER*     1 SEX
CHARACTER*     2 AGE
CHARACTER*    10 FAMSTA
CHARACTER*     2 NUMNTS
CHARACTER*    51 PERSON
CHARACTER*    15 PDES
CHARACTER*    51 PERNEL
```

Note:
Synonyms are assumed to be defined in the data dictionary as shown in Appendix B and truncation is assumed to occur in the middle of the word. (The maximum length of names depends on the operating system.)

Note:
The field PERNEL encompasses all other fields and is the equivalent of the record buffer in Ada, COBOL and PL/I.

The structure of the PL/I record buffer is as follows:

```
DCL 1  PERSONNEL,
      2 RECORD_BUFPERS_1 UNAL,
       3 PERSON,
        4 NAME                    CHAR (20),
        4 FIRST_NAME              CHAR (15),
        4 INITIAL                 CHAR (1),
        4 P_DES,
         5 SEX                    CHAR (1),
         5 AGE                    PIC ’(1)99’,
         5 FAMILY_STATUS          CHAR (10),
         5 NUMBER_OF_DEPENDENTS   PIC ’(1)99’,
      2 ISN                       FIXED BIN(31),
      2 QUANTITY                  FIXED BIN(31),
      2 RESPONSE_CODE             FIXED BIN(15),
       RECORD_BUFPERS CHAR(51) BASED (ADDR(RECORD_BUFPERS_1));
```

Any field within a group may also be specified as a single field name.

Note:
The level-2 name generated for the record buffer includes the cursor-name, if one was specified. The COBOL example shows a record buffer that was generated from an Adabas Native SQL statement without a cursor-name; the Ada and PL/I examples show a record buffer that was generated from an Adabas Native SQL statement with the cursor-name PERS.

Multiple-Value Fields

A multiple-value (MU) field is specified as a single field name; Adabas Native SQL takes the number of occurrences from the data dictionary. If the number of occurrences is specified as zero in the data dictionary, then Adabas Native SQL will declare 191 occurrences of the field. It is therefore strongly recommended that the number of occurrences be correctly specified in the data dictionary.

A single occurrence or a range of occurrences may optionally be specified within parentheses. The upper limit of the range or the number of the occurrence must not be greater than the number of occurrences as specified in the data dictionary, otherwise it will be ignored and a warning message will be printed. The valid formats are:

mu

mu(i)

mu(:var)

mu(i-j)

mu(LAST)

mu(i-LAST) (only at the end of the SELECT list)

where *mu* denotes the name of the multiple field; *i* and *j* denote integer constants; and *var* denotes the name of an integer variable. In Ada, *var* must be defined as "STRING(1..5)". In FORTRAN, *var* must be defined as "CHARACTER*5" and should contain a 5-digit number. LAST may be specified as the occurrence of an MU field to indicate that the last occurrence is to be read. For MU fields it is also possible to specify (*i*-LAST) at the end of the SELECT list to indicate a range of occurrences, from the occurrence with number *i* through to the last occurrence.

If a multiple-value field is referenced in the WHERE clause of a data retrieval statement, the only valid format is:

mu

If a single occurrence or a range not starting from 1 is specified, the name in the record buffer will be followed by a "-" or "_" and the number of the occurrence or the range.

Example:

```
SELECT OIL-CREDIT(1-5), OIL-CREDIT(7), OIL-CREDIT(9-10)
FROM FINANCE
```

The structure of the Ada record buffer is as follows:

```
type OIL_CREDITPERS is array (INTEGER range <>)
                      of STRING (1..7);
type OIL_CREDIT_9_10PERS is array (INTEGER range <>)
                            of STRING (1..7);

type RECORD_BUFPERS is
   record
      OIL_CREDIT      : OIL_CREDITPERS (1..5);
      OIL_CREDIT_7    : STRING (1..7);
      OIL_CREDIT_9_10 : OIL_CREDIT_9_10PERS (1..2);
      ISN             : INTEGER;
      QUANTITY        : INTEGER;
      RESPONSE_CODE   : SHORT_INTEGER;
   end record;
FINANCE : RECORD_BUFFERS;
```

The structure of the COBOL record buffer is as follows:

```
01  FINANCE.
    02 RECORD-BUFPERS.
     03 OIL-CREDIT      PIC X(7) OCCURS 5.
     03 OIL-CREDIT-7    PIC X(7).
     03 OIL-CREDIT-9-10 PIC X(7) OCCURS 2.
    02 ISN              PIC 9(9) COMP.
    02 QUANTITY         PIC 9(9) COMP.
    02 RESPONSE-CODE    PIC 9(4) COMP.
```

The FORTRAN equivalent is as follows:

```
CHARACTER*     7 OCRE   (00005)
CHARACTER*     7 OCRE7
CHARACTER*     7 OCR910 (00002)
CHARACTER*    56 FINNCE
```

Note:
Synonyms are assumed to be defined in the data dictionary as shown in Appendix B and truncation is assumed to occur in the middle of the word. (The maximum length of names depends on the operating system.)

Note:
The field FINNCE encompasses all other fields and is the equivalent of the record buffer in Ada, COBOL and PL/I.

The structure of the PL/I record buffer is as follows:

```
DCL 1  FINANCE,
      2 RECORD_BUFPERS_1 UNAL,
       3 OIL_CREDIT      (5) CHAR (7),
       3 OIL_CREDIT_7        CHAR (7),
       3 OIL_CREDIT_9_10 (2) CHAR (7),
      2 ISN                  FIXED BIN(31),
      2 QUANTITY             FIXED BIN(31),
      2 RESPONSE_CODE        FIXED BIN(15),
       RECORD_BUFPERS CHAR(56) BASED (ADDR(RECORD_BUFPERS_1));
```

If the range is not explicitly specified, the default range is from the first occurrence up to the number specified in the data dictionary file (or 191 if the number of occurrences is not specified in the data dictionary).

In conjunction with multiple-value fields, you may additionally code *mu*(COUNT), i.e., the field name followed by the keyword COUNT in parentheses. This causes Adabas Native SQL to generate a special field in which Adabas stores the actual number of occurrences in the record. The field is two bytes long and has the following binary format:

- SHORT_INTEGER in ADA;
- PIC S9(4) COMP in COBOL;
- INTEGER*2 in FORTRAN;
- FIXED BIN(15,0) in PL/I.

The name generated for the COUNT field is the same as the name of the multiple-value field, preceded by:

- "C_" in ADA;
- "C-" in COBOL;
- "C" in FORTRAN;
- "C_" in PL/I.

A count field is also generated if a count field is defined in a Predict field maintenance function. This is particularly useful in conjunction with the Adabas Native SQL `SELECT *` statement. A count field is never generated for a multiple-value field within a periodic group.

Example:

```
SELECT OIL-CREDIT, OIL-CREDIT(COUNT)
FROM FINANCE
```

The structure of the Ada record buffer is as follows:

```
   type OIL_CREDITPERS is array (INTEGER range <>)
                          of STRING (1..7);
   type RECORD_BUFPERS is
     record
        OIL_CREDIT    : OIL_CREDITPERS (1..191);
        C_OIL_CREDIT  : SHORT_INTEGER;
        ISN           : INTEGER;
        QUANTITY      : INTEGER;
        RESPONSE_CODE : SHORT_INTEGER;
    end record;
FINANCE: RECORD_BUFPERS;
```

The structure of the COBOL record buffer is as follows:

```
01  FINANCE.
    02 RECORD-BUFPERS.
     03 OIL-CREDIT   PIC X(7) OCCURS 191.
     03 C-OIL-CREDIT PIC S9(4) COMP.
    02 ISN           PIC 9(9) COMP.
    02 QUANTITY      PIC 9(9) COMP.
    02 RESPONSE-CODE PIC 9(4) COMP.
```

The FORTRAN equivalent is as follows:

```
CHARACTER*     7 OCRE   (00191)
INTEGER*       2 COCRE
CHARACTER*  1340 FINNCE
```

Note:
Synonyms are assumed to be defined in the data dictionary as shown in Appendix B and truncation is assumed to occur in the middle of the word. (The maximum length of names depends on the operating system.)

Note:
The field FINNCE encompasses all other fields and is the equivalent of the record buffer in Ada, COBOL and PL/I.

The structure of the PL/I record buffer is as follows:

```
DCL 1  FINANCE,
      2 RECORD_BUFPERS_1 UNAL,
       3 OIL_CREDIT   (191) CHAR (7),
       3 C_OIL_CREDIT       FIXED BIN(15,0),
      2 ISN                 FIXED BIN(31),
      2 QUANTITY            FIXED BIN(31),
      2 RESPONSE_CODE       FIXED BIN(15),
       RECORD_BUFPERS CHAR(1339) BASED (ADDR(RECORD_BUFPERS_1));
```

Periodic Groups

A periodic group (PE) consists of up to 65000 occurrences of a group. The default number of occurrences remains 99, as in the previous version. Adabas Native SQL automatically generates definitions of all fields within the periodic group, using the full field names as defined in the data dictionary, or the Ada, COBOL, FORTRAN or PL/I synonyms if present. You may limit the number of occurrences as for multiple value fields. A COUNT field containing the number of occurrences of the periodic group may be generated by coding *pe*(COUNT) or by defining a PE count field with a Predict field maintenance function. Valid formats:

```
pe

pe(i)

pe(:var)

pe(i-j)
```

where *pe* denotes the name of the periodic group; *i* and *j* denote integer constants; and var denotes the name of an integer variable. In Ada, var must be defined as "STRING(1..5)". In FORTRAN, *var* must be defined as "CHARACTER*5" and should contain a 5-digit number.

If a periodic group is referenced in the WHERE clause of a data retrieval statement, the valid formats are:

```
pe

pe(i)
```

Suffixes defining a single occurrence or a range of occurrences not starting from 1 will be added to all fields within the periodic group. A range starting from the first occurrence is not given a suffix.

If you do not need all the fields within the periodic group, you may request individual fields, which are treated as multiple-value fields, except that you may not request the COUNT of such a field, but only the COUNT of the periodic group as a whole.

For COBOL and PL/I, Adabas Native SQL supports the `GROUP STRUCT` attribute which can be defined in the data dictionary for periodic groups. Correct use of this attribute can result in a significantly shorter Adabas format buffer. For more information see *Defining More Attributes of Fields, 3GL Specification* in section *Field* of Chapter *Predefined Object Types* of the Predict Reference Manual.

Note:
(for Ada and FORTRAN users): Periodic groups will always be generated with GROUP STRUCT = N, and no consideration will be given to the Predict definition.

Example:

```
SELECT MAJOR-CREDIT(1), MAJOR-CREDIT(3-5), MAJOR-CREDIT(7),
       MAJOR-CREDIT(COUNT)
FROM FINANCE
```

The structure of the Ada record buffer is as follows:

```
   type CREDIT_CARD_3_5PERS is array (INTEGER range <>)
                            of STRING (1..18);
   type CREDIT_LIMIT_3_5PERS is array (INTEGER range <>)
                             of STRING (1..4);
   type CURRENT_BALANCE_3_5PERS is array (INTEGER range <>)
                                of STRING (1..4);
   type RECORD_BUFPERS is
      record
        CREDIT_CARD_1       : STRING (1..18);
        CREDIT_LIMIT_1      : STRING (1..4);
        CURRENT_BALANCE_1   : STRING (1..4);
        CREDIT_CARD_3_5     : CREDIT_CARD_3_5PERS (1..3);
        CREDIT_LIMIT_3_5    : CREDIT_LIMIT_3_5PERS (1..3);
        CURRENT_BALANCE_3_5 : CURRENT_BALANCE_3_5PERS (1..3);
        CREDIT_CARD_7       : STRING (1..18);
        CREDIT_LIMIT_7      : STRING (1..4);
        CURRENT_BALANCE_7   : STRING (1..4);
        C_MAJOR_CREDIT      : SHORT_INTEGER;
        ISN                 : INTEGER;
        QUANTITY            : INTEGER;
        RESPONSE_CODE       : SHORT_INTEGER;
      end record;
FINANCE: RECORD_BUFPERS;
```

The structure of the COBOL record buffer is as follows:

```
01  FINANCE.
    02 RECORD-BUFPERS.
     03 MAJOR-CREDIT-1.
      04 CREDIT-CARD-1        PIC X(18).
      04 CREDIT-LIMIT-1       PIC 9(4).
      04 CURRENT-BALANCE-1    PIC 9(4).
     03 G-MAJOR-CREDIT-3-5.
      04 MAJOR-CREDIT-3-5               OCCURS 3.
       05 CREDIT-CARD-3-5     PIC X(18).
       05 CREDIT-LIMIT-3-5    PIC 9(4).
       05 CURRENT-BALANCE-3-5 PIC 9(4).
     03 MAJOR-CREDIT-7.
      04 CREDIT-CARD-7        PIC X(18).
      04 CREDIT-LIMIT-7       PIC 9(4).
      04 CURRENT-BALANCE-7    PIC 9(4).
     03 C-MAJOR-CREDIT        PIC S9(4) COMP.
    02 ISN                    PIC 9(9) COMP.
    02 QUANTITY               PIC 9(9) COMP.
    02 RESPONSE-CODE          PIC 9(4) COMP.
```

The FORTRAN equivalent is as follows:

```
CHARACTER*    18 CCARD1
CHARACTER*     4 CLIM1
CHARACTER*     4 CBAL1
CHARACTER*    26 MAJIT1
CHARACTER*    18 CCAD35(00003)
```

```
CHARACTER*     4 CLIM35(00003)
CHARACTER*     4 CBAL35(00003)
CHARACTER*    78 MAJT35
CHARACTER*    18 CCARD7
CHARACTER*     4 CLIM7
CHARACTER*     4 CBAL7
CHARACTER*    26 MAJIT7
INTEGER*       2 CMADIT
CHARACTER*   132 FINNCE
```

Note:
Synonyms are assumed to be defined in the data dictionary as shown in Appendix B and truncation is assumed to occur in the middle of the word. (The maximum length of names depends on the operating system.)

Note:
The field FINNCE encompasses all other fields and is the equivalent of the record buffer in Ada, COBOL and PL/I.

The structure of the PL/I record buffer is as follows:

```
DCL 1  FINANCE,
      2 RECORD_BUFPERS_1 UNAL,
       3 MAJOR_CREDIT_1,
        4 CREDIT_CARD_1          CHAR(18),
        4 CREDIT_LIMIT_1         PIC '(3)99',
        4 CURRENT_BALANCE_1      PIC '(3)99',
       3 G_MAJOR_CREDIT_3_5,
        4 MAJOR_CREDIT_3_5   (3),
         5 CREDIT_CARD_3_5       CHAR(18),
         5 CREDIT_LIMIT_3_5      PIC '(3)99',
         5 CURRENT_BALANCE_3_5   PIC '(3)99',
       3 MAJOR_CREDIT_7,
        4 CREDIT_CARD_7          CHAR(18),
        4 CREDIT_LIMIT_7         PIC '(3)99',
        4 CURRENT_BALANCE_7      PIC '(3)99',
       3 C_MAJOR_CREDIT          FIXED BIN(15,0),
      2 ISN                      FIXED BIN(31),
      2 QUANTITY                 FIXED BIN(31),
      2 RESPONSE_CODE            FIXED BIN(15),
       RECORD_BUFPERS CHAR(132) BASED(ADDR(RECORD_BUFPERS_1));
```

Multiple-Value Fields within Periodic Groups

Adabas Native SQL supports multiple-value fields that occur within periodic groups. If the number of occurrences is not specified, the number of occurrences is taken from the data dictionary. If the number of occurrences is not explicitly specified, or if the index is variable, the occurrence number is not appended as a suffix to the field name.

Reference to elements of such a field is made as follows:

mp

mp(i(k))

mp(i(k-l))

```
mp(i-j(k))

mp(i-j(k-l))

mp(:ivar(k))

mp(:ivar(k-l))

mp(i(:kvar))

mp(i-j(:kvar))

mp(:ivar(:kvar))

mp(LAST)

mp(LAST(LAST))

mp(i(k-LAST))      (only at the end of the SELECT list)
```

mp denotes the name of the multiple-value field. *i*, *i-j* and *ivar* indicate which group or groups are required. *k*, *k-l* and kvar indicate which occurrence or occurrences of the multiple-value field are required. *i*, *j*, *k* and *l* denote integer constants. *j* must be greater than *i*, and both must be in the range 1..191. *l* must be greater than *k*, and both must be in the range 1..191. *ivar* and *kvar* denote the names of integer variables. LAST means the last occurrence.

If a multiple-value field within a periodic group is referenced in the WHERE clause of a data retrieval statement, the only valid format is:

```
mp
```

Counter fields can also be generated for multiple-value fields occurring within periodic groups. *mp*(COUNT1) generates a counter field containing the number of occurrences of the multiple- value field *mp* in the first occurrence of the periodic group, *mp*(COUNT1-3) generates counter fields for the multiple-value field *mp* in each of the first three occurrences of the periodic group, and *mp*(COUNTLAST) generates a counter field for the multiple-value field in the last occurrence of the periodic group. The names of the counter fields are:

ADA	**COBOL**	**FORTRAN**	**PL/I**
C_*mp*_1	C-*mp*-1	C*mp*1	C_*mp*_1
C_*mp*_2	C-*mp*-2	C*mp*2	C_*mp*_2
C_*mp*_3	C-*mp*-3	C*mp*2	C_*mp*_3

Example:

```
SELECT INSURANCE-COMPANY(2-4(6-8))
FROM FINANCE
```

The structure of the Ada record buffer is as follows:

```
type INSURANCE_COMPANY_6_8PERS is array (INTEGER range <>,
                                         INTEGER range <>)
                              of STRING (1..25);
type RECORD_BUFPERS is
  record
    INSURANCE_COMPANY_6_8 : INSURANCE_COMPANY_6_8PERS (1..3, 1..3);
    ISN                   : INTEGER;
    QUANTITY              : INTEGER;
    RESPONSE_CODE         : SHORT_INTEGER;
  end record;
FINANCE: RECORD_BUFPERS;
```

The structure of the COBOL record buffer is as follows:

```
01  FINANCE.
    02 RECORD-BUFPERS.
     03 A-INSURANCE-COMPANY-2-4           OCCURS 3.
      04 INSURANCE-COMPANY-6-8  PIC X(25) OCCURS 3.
    02 ISN                      PIC 9(9) COMP.
    02 QUANTITY                 PIC 9(9) COMP.
    02 RESPONSE-CODE            PIC 9(4) COMP.
```

The FORTRAN equivalent is as follows:

```
CHARACTER*    25 INCM68(00003 , 00003)
CHARACTER*   225 FINNCE
```

Note:
Synonyms are assumed to be defined in the data dictionary as shown in Appendix B and truncation is assumed to occur in the middle of the word. (The maximum length of names depends on the operating system.)

Note:
The field FINNCE encompasses all other fields and is the equivalent of the record buffer in Ada, COBOL and PL/I.

The structure of the PL/I record buffer is as follows:

```
DCL 1  FINANCE,
      2 RECORD_BUFPERS_1 UNAL,
       3 A_INSURANCE_COMPANY_2_4     (3),
        4 INSURANCE_COMPANY_6_8      (3) CHAR(25),
      2 ISN                              FIXED BIN(31),
      2 QUANTITY                         FIXED BIN(31),
      2 RESPONSE_CODE                    FIXED BIN(15),
       RECORD_BUFPERS CHAR(225) BASED(ADDR(RECORD_BUFPERS_1));
```

Additional Fields in the Record Buffers (Ada, COBOL, PL/I)

If a field is specified in the SELECT clause, and Predict contains redefinitions for this field, then the redefined fields are also included in the record buffer. The prefix and suffix are added to the field names and the result is truncated if necessary. (Ada does not support redefinition.)

Unless the global parameter `ABORT .` is specified, Adabas Native SQL appends three fields to each record buffer. A record buffer containing these three fields is also generated for DELETE statements, although no database fields are generated. The names of the fields are shown in the tables below. They may only be used in conjunction with an adequate file name.

If the global parameter `ABORT .` is specified, these three fields are generated as global data and they have the names SQLISN, SQLQTY and SQLRSP, as used in FORTRAN programs. Since no record buffers are ever generated for FORTRAN, the field names are always global to the program.

The ISN variable is a 4-byte binary field in which Adabas returns the ISN (internal sequence number) of the (first) record found or read or, in the case of a HISTOGRAM command where the descriptor is in a periodic group, the number of the current occurrence. The ISN variable is defined as:

Language	Variable Name*	Format
ADA	ISN	INTEGER
COBOL	ISN	PIC 9(9) COMP
PL/I	ISN	FIXED BIN (31)

* The variable name is SQLISN if the global parameter `ABORT .` is coded. See description of the ABORT parameter for more information.

The QUANTITY variable is a 4-byte binary field which, when used in conjunction with a COMPARE, FIND, FIND COUPLED or SORT statement, is available after executing the OPEN statement. It returns the number of ISNs in the ISN list, or the number of ISNs in the ISN buffer. When used in conjunction with a HISTOGRAM statement, the quantity variable, which is available after executing the FETCH statement, returns the number of records that contain the specified descriptor value. (The quantity variable is not available in conjunction with READ statements.) The quantity variable is defined as:

Language	Variable Name*	Format
ADA	QUANTITY	INTEGER
COBOL	QUANTITY	PIC 9(9) COMP
PL/I	QUANTITY	FIXED BIN (31)

* The variable name is SQLQTY if the global parameter `ABORT .` is coded. See description of the ABORT parameter for more information.

The RESPONSE_CODE (Ada), RESPONSE-CODE (COBOL) or RESPONSE_CODE (PL/I) variable is a 2-byte binary field in which Adabas returns the response code after execution of the command. The response code variable is defined as:

Language	Variable Name*	Format
ADA	RESPONSE_CODE	SHORT_INTEGER
COBOL	RESPONSE-CODE	PIC 9(4) COMP
PL/I	RESPONSE_CODE	FIXED BIN (15)

* The variable name is SQLRSP if the global parameter `ABORT .` is coded.

See *Response Code Interpretation* and the description of the ABORT parameter for more information.

Additional Fields in FORTRAN Programs

Adabas Native SQL enters values in three global variables after each SQL statement. These variables contain only the values generated by the last command and will be changed when a new command is issued.

The ISN variable is a 4-byte binary field in which Adabas returns the ISN (internal sequence number) of the (first) record found or read or, in the case of a HISTOGRAM command where the descriptor is in a periodic group, the number of the current occurrence.

Language	Variable Name	Format
FORTRAN	SQLISN	INTEGER*4

The QUANTITY variable is a 4-byte binary field which, when used in conjunction with a COMPARE, FIND, FIND COUPLED or SORT statement, is available after executing the OPEN statement. It returns the number of ISNs in the ISN list, or the number of ISNs in the ISN buffer. When used in conjunction with a HISTOGRAM statement, the quantity variable, which is available after executing the FETCH statement, returns the number of records that contain the specified descriptor value. (The quantity variable is not available in conjunction with READ statements.)

Language	Variable Name	Format
FORTRAN	SQLQTY	INTEGER*4

The response code variable is a 2-byte binary field in which Adabas returns the response code after execution of the command.

Language	Variable Name	Format
FORTRAN	SQLRSP	INTEGER*2

See *Response Code Interpretation* and the description of the ABORT parameter for more information.

If you want to use for example the response codes returned by more than one statement, then you must save each response code before new SQL statements are executed.

End-of-File Flag (ADACODE, SQLCOD)

The ADACODE (Ada, COBOL, DEC FORTRAN and PL/I) or SQLCOD (IBM FORTRAN) variable is a 2-byte binary field in which Adabas Native SQL returns an end-of-file flag. The value 3 in this field indicates that end-of-file was detected in a sequential read command, or end-of-list after reading all the records found by a search statement. It is defined as:

Language	Variable Name	Format
ADA	ADACODE	SHORT_INTEGER
COBOL	ADACODE	PIC 9(4) COMP
FORTRAN	SQLCOD	INTEGER*2
FORTRAN/VMS	ADACODE	INTEGER*2
PL/I	ADACODE	FIXED BIN (15)

Response Code Interpretation

The Adabas response code is a code that is returned to the caller after every Adabas command. It is stored in a variable called RESPONSE-CODE (COBOL) or RESPONSE_CODE (Ada and PL/I) in the record buffer of the command that was executed, or in the global variable SQLRSP (FORTRAN). A value of zero returned in this variable indicates that the Adabas Native SQL statement has been executed successfully. A non-zero value (other than 3, which denotes end-of-file) indicates that an error occurred. In this case, the statement has not been executed. Each value is associated with a distinct type of error, as shown in the list below.

Adabas Native SQL automatically calls an error-checking routine after each Adabas command if the response code is non-zero. Software AG supplies default routines which check and interpret the response code. If the response code has a value other than 3, the routine prints out the appropriate error message, the contents of the Adabas control block and the line number of the erroneous statement in the source program, calls an appropriate trace module, issues a `backout transaction` (ROLLBACK WORK) command, closes the database (DBCLOSE), and finally terminates the program.

Language	Default Abort Module	Default Trace Module
ADA	RESPF	PRTRAC
COBOL	RESPINT	PRTRACE
FORTRAN	RESPF	PRTRAC
PL/I	RESPINT	PRTRACE

In many cases, the action described above may be all that is required. However, if the action taken by the standard routine is inappropriate or insufficient, the ABORT parameter can be used to specify that a user-defined error handling routine with a different name should be called instead. The data administrator will know whether alternative error handling routines are available at your installation.

See also the description of the ABORT parameter.

Response Codes

The response code is returned in the variable RESPONSE_CODE (Ada), RESPONSE-CODE (COBOL), SQLRSP (FORTRAN) or RESPONSE_ CODE (PL/I) that is attached to every record buffer. The normal response code (success) is 0.

If the following response code occurs and the error handling routine is that shown in the table above, control will be returned to the user program directly following the statement that caused the response code.

Response code	Meaning
3	Response Code 3 (which is also signaled in the variable ADACODE (Ada, COBOL or PL/I) or SQLCOD (FORTRAN)) indicates that end-of-file was detected in a sequential read command, or end-of-list after reading all the records found by a search statement.

The following response codes may also occur during normal operation. If a user-written error handling routine is called, it should take appropriate action for all response codes that might occur. This might include printing an error message and/or returning to the application program. The standard error handling routines *RESPINT* and *RESPF* supplied by Software AG can be used as a model when writing this routine.

Response code	**Meaning**
1	The ISN list is too big to be sorted.
9	A partially-completed transaction has been automatically backed-out, possibly as the result of a timeout (for programs that use ET-mode). Note that Adabas may release the command-ID when Response Code 9 occurs. ISN lists, hold queue entries and user data (see also the CHECKPOINT, COMMIT WORK, CONNECT, DBCLOSE and READ USERDATA statements) are no longer accessible.
17	Invalid file number. A file required by the program could not be found in the database
19	An attempt has been made to update a file that was opened for access only
41	Adabas has detected an error in the format buffer. This can be caused by an incorrect data field definition in Predict.
48	The user-ID specified in the CONNECT statement is already in use; or the mode of usage specified for a file in the CONNECT statement conflicts with the file's current usage.
98	A descriptor value in a record to be INSERTed or UPDATEd exists already in the file and the file has the 'unique descriptor' attribute (VAX response code).
113	A READ ISN statement without the SEQUENCE option was issued and Adabas could not find a record having the specified ISN; or a READ ISN statement attempted to read a record and the 'security by value' check failed. It can also indicate that an INSERT statement using the `WHERE ISN=n` clause specified an ISN that was already present in the file.
144	An UPDATE or DELETE statement was issued but the relevant record was not in hold status for the program that issued the statement.
145	The program attempted to hold a record that is already being held by another user. This code may be returned if the HOLD RETURN option is used.
148	The Adabas nucleus is not available.
198	A descriptor value in a record to be INSERTed or UPDATEd exists already in the file and the file has the 'unique descriptor' attribute.

See Chapter *Adabas Response Codes* in the *Adabas Messages and Codes Manual* for more information.

Host Variables

Host variables are normal program variables that are also used in Adabas Native SQL statements. They are declared using normal Ada, COBOL, FORTRAN or PL/I statements. When used in an Adabas Native SQL statement, the name of each host variable must be immediately preceded by a colon (":"), for example ":NAME".

ISN Lists and the ISN Buffer

The abbreviation ISN occurs frequently in this manual. It stands for Internal Sequence Number: a reference number that identifies each record uniquely within an Adabas file. Each new record created by the INSERT statement must have an ISN. If you do not allocate the ISN explicitly, it is assigned automatically by Adabas. When allocating ISNs, care should be taken that each ISN is unique and that no ISN that exceeds the MAXISN parameter is specified.

When a FIND statement finds more than one record in the file, Adabas makes a list of the ISNs of these records and returns this ISN list as the result of the FIND operation.

You have the option of providing an ISN buffer, whose size is specified by the ISNSIZE parameter either in the global OPTIONS parameter or in each individual Adabas Native SQL statement. If an ISN buffer of adequate size is provided, Adabas stores the ISN list in this buffer. If an ISN buffer is not provided, or if it is too small to contain the ISN list created by a particular FIND statement, then the excess ISNs are automatically written to the Adabas workfile. They are then read from the ISN buffer and/or from the workfile and returned to the user one by one each time a statement (for example, FETCH) that requires an ISN is executed.

In general, programs run more efficiently if the ISN buffer is large enough to contain the entire ISN list. However, if the ISN buffer has to be made smaller, the program will continue to run exactly as before; the process of buffering excess ISNs in the Adabas workfile is completely transparent to the user.

The ISN buffer cannot be used if Adabas security by value is in effect, or in CICS or UTM programs that use the Adabas Native SQL statements SAVE and RESTORE.

HOLD Logic

The HOLD option can be used with all Adabas Native SQL data retrieval statements except HISTOGRAM to place the record in hold status. A record in hold status is prevented from being updated by other users until it is explicitly released by issuing a COMMIT WORK, ROLLBACK WORK or RELEASE statement. This avoids the conflict that would arise if two or more users attempted to update one record simultaneously.

RETURN Option

The presence or absence of the RETURN option determines Adabas's response if the record to be accessed is currently being held by another user.

If HOLD is used without the RETURN option and an attempt is made to access a record held by another user, the program is suspended until the record is released by the other user.

If HOLD is used with the RETURN option and an attempt is made to access a record held by another user, Adabas returns Response Code 145 to the user program. If the response code interpretation routine as supplied by Software AG is being used, an error message is printed and the program ABENDs. If some other action is required, an alternative routine that checks for this response code and takes appropriate action must be supplied (see also the description of the ABORT parameter). The response code is returned in the variable RESPONSE_CODE (Ada), RESPONSE-CODE (COBOL) or RESPONSE_CODE (PL/I), which is attached to every record buffer, or in the global variable SQLRSP (FORTRAN).

See section *Competitive Updating* in the Adabas Command Reference Manual for more information on Adabas hold logic.

Security Options

Adabas offers the following facilities to prevent unauthorized users from accessing or updating confidential data:

- Password protection
- Ciphering
- Security by value.

Password Protection

Password protection permits only those database operations that cite the correct password. Adabas commands that include an incorrect password, or no password at all, are rejected. Furthermore, access and update security levels are associated with each password. Whenever a database operation is executed, Adabas checks that the security level associated with the password equals or exceeds the security level of the database, both at the file level and at the field level. Password protection therefore provides a very flexible mechanism for controlling the degree of access individual computer users can exercise.

Ciphering

If a file is ciphered, the data are stored on disk in an encrypted format that is incomprehensible to any user who does not know the correct cipher key. Adabas uses the cipher key in conjunction with a special decryption algorithm to reconstruct the original data. Cipher protection offers a very high level of security against unauthorized efforts to read data from a database. Conversely, a file update made with a wrong cipher key is conspicuous because the decryption algorithm converts the data into a meaningless jumble when a legitimate user tries to read them.

Further details of the password and data encryption security facilities are given in the section *Security Planning* in the *Adabas DBA Reference Manual*.

Security by Value

The third security option Adabas offers is security by value. Using this facility, access to records is controlled by the values contained in specified fields. For example, a user may be forbidden from accessing records in the PERSONNEL file that have a value in the SALARY field exceeding 6000.

The ISNSIZE option cannot be used when processing files that are protected by this feature. See page for more information.

See the *Adabas Security Manual* for more information. Note that this manual is only sent to DBAs on written application.

Consult your DBA before writing programs that access files protected by any of the mechanisms described in this section.

Record Buffer - ADA

The fields generated in the record buffers in Ada programs have the clauses shown in the table below:

Predict Format	**Predict Length**	**Ada clause**	**Observations**
A	*nnn*	STRING (1..*nnn*)	
B or I	1	SHORT_SHORT_INTEGER	VMS only
B or I	2	SHORT_INTEGER	
B or I	4	INTEGER	
F	4	FLOAT	
F	8	LONG_FLOAT	VMS only
N or U	*nn.m*	STRING (1..*nn+m*)	
P	*nn.m*	STRING (1..*y*)	
L		BOOLEAN	
D		STRING (1..4)	
T		STRING (1..7)	
Counter fields	SHORT_INTEGER		

Note:
Numeric fields are transformed into character fields; therefore, whenever these fields are initialized and whenever values are assigned to these fields, the values must be filled with leading zeros, for example "0001".

Note:
$y = (nn+m+1) / 2$

Record Buffer - COBOL

The fields generated in the record buffers in COBOL programs have the clauses shown in the table below:

Predict Format	**Predict Length**	**COBOL clause**	**Observations**
A	nnn	PIC X(nnn)	
B or I	2	PIC S9(4) COMP	
B or I	4	PIC S9(9) COMP	
B or I	8	PIC S9(18) COMP	
F	4	COMP-1	
F	8	COMP-2	
N or U	nn.m	PIC 9(nn)V9(m)	In any of these fields, nn+m may not exceed 18, and if m=0 the term V9(m) is omitted
NS or US	nn.m	PIC S9(nn)V9(m)	
P	nn.m	PIC 9(nn)V9(m)COMP-3	
PS	nn.m	PIC S9(nn)V9(m)COMP-3	
L		PIC X	
D		PIC 9(7) COMP-3	
T		PIC 9(13) COMP-3	
Counter fields		PIC S9(4) COMP	

An automatically generated counter field has the clause PIC S9(4) COMP.

A numeric or binary format field with a length not included in the table above is treated in COBOL as an alphanumeric format field

Packed fields in COBOL/II under operating system BS2000/OSD are generated as "PACKED DECIMAL" instead of "COMP-3".

No alignment is performed.

Fields in FORTRAN

The fields generated in FORTRAN programs have the clauses shown in the table below:

Predict		FORTRAN Clause	Compiler	
Format	**Length**			**Alignment assuming word length=4**
A	nnn	CHARACTER*nnn	any	
B or I	1	LOGICAL*1	IBM, Siemens, VMS	
B or I	2	INTEGER*2	IBM, Siemens, VMS	half-word boundary
B or I	4	INTEGER*4	IBM, Siemens, VMS	word boundary
B or I	8	INTEGER*8	Siemens	double-word boundary
B or I	8	CHARACTER*8	IBM, VMS	
F	4	REAL*4	IBM, Siemens, VMS	word boundary
F	8	REAL*8	IBM, Siemens, VMS	double-word boundary
N or NS, U or US	nn.m	CHARACTER*x where x=nn+m	any	
P or PS	nn.m	CHARACTER*y where y=(nn+m+1)/2	any	
L		LOGICAL*1	any	
D		CHARACTER*4	any	
T		CHARACTER*7	any	

- If generated for IBM, Siemens or VMS compilers: Any file number field, length fields and automatically generated counter fields have the clause INTEGER*2.

Note:
Numeric fields are transformed into character fields; therefore, whenever these fields are initialized and whenever values are assigned to these fields, the values must be filled with leading zeros, for example "0001".

Record Buffer - PL/I

The fields generated in the record buffers in PL/I programs have the clauses shown in the table below:

Fields in the PL/I include code have a PL/I clause determined by the length and format of the corresponding Predict field object, as shown in the table below where s is the numeric sign whose content (*T*, *I*, or *9R*) and position (left or right) are defined in the PL/I generation defaults; *nn+m* must not exceed 15; and if *m* is zero, *V(m)9* is omitted.

Predict		**PL/I clause**	**Observations**
Format	**Length**		
A	nnn	CHAR (nnn)	
B	1	FIXED BIN(7)	VMS only
B or I	2	FIXED BIN (15,0)	
B or I	4	FIXED BIN (31,0)	
F	4	FLOAT DEC (6)	
F	8	FLOAT DEC (16)	
N or U	nn.m	PIC ’(nn)9V(m)9’	
NS or US	nn.m	PIC ’s(nn-1)9V(m)9’ or PIC ’(nn)9V(m-1)9s’	
P or PS	nn.m	FIXED DEC (nn+m,m)	
L		BIT (8)	
D		FIXED DEC (7,0)	
T		FIXED DEC(13,0)	
Counter fields		FIXED BIN (15,0)	

A numeric or binary format field with a length not included in the table above is treated in PL/I as an alphanumeric format field.

Date and Time Conversion Routines

The following routines are delivered with this version of Adabas Native SQL and can be used in the application:

- SQTODATE
- SQFRDATE
- SQTOTIME
- SQFRTIME

SQTODATE

This module accepts two parameters:

- N-DATE (N8) in format DDMMYYYY
- DATE (D)

It converts the first parameter into a format D number and returns it in the second parameter.

SQFRDATE

This module accepts two parameters:

- N-DATE (N8) in format DDMMYYYY
- DATE (D)

It converts the second parameter, which is a format D number, into a numeric date and returns it in the first parameter.

SQTOTIME

This module accepts three parameters:

- N-DATE (N8) in format DDMMYYYY
- N-TIME (N7) in format HHMMSSS
- TIME (T)

It converts the first and second parameters into a format T number and returns it in the third parameter.

SQFRTIME

This module accepts three parameters:

- N-DATE (N8) in format DDMMYYYY
- N-TIME (N7) in format HHMMSSS
- TIME (T)

It converts the third parameter, which is a format T number, into a numeric date and numeric time and returns them in the first and second parameters.

Support of Distributed Data Structures

Adabas Native SQL supports distributed data structures by the DBID or AUTODBID clauses in Adabas Native SQL statements, or the Global OPTIONS parameters AUTODBID-ALL , AUTODBID-ATM , AUTODBID and DBID. These clauses put the DBID number defined in Predict in the control block.

- The Global Parameters NETWORK and VIRTUAL-MACHINE

The Global Parameters NETWORK and VIRTUAL-MACHINE

These global parameters are mandatory if more than one network is defined in Predict.

These parameters define the network and virtual machine in which the program is to run. Adabas Native SQL checks that the network and virtual machine exist in Predict and that the virtual machine is linked as a child object to the network.

For every database used (DBID, AUTODBID, AUTODBID-ATM and AUTODBID-ALL clauses) Adabas Native SQL checks the following:

- that if the database is defined as local, it is linked to the current virtual machine,
- that if the database is defined as isolated, it is linked (via the current virtual machine) to the current network.

Note:
In this section, the terms *current network* and *current virtual machine* are used to describe the network and virtual machine specified with the global parameters NETWORK and VIRTUAL-MACHINE respectively.

The Distribution handling

The distribution is handled by the application programmer. If the program uses the DBID, AUTODBID, AUTODBID-ATM and AUTODBID-ALL, Adabas Native SQL performs the following additional checks:

- If one of the DBID clauses is used, the Run Mode parameter of the corresponding Predict database object must be *I* (isolated) or *L* (local), otherwise an error message is given.
- If the database is *local*, Adabas Native SQL checks that it is linked to the current virtual machine.
- If the database is *isolated*, Adabas Native SQL checks that it is linked to the current network.

After checking the database, Adabas Native SQL checks the physical link between the file and the database. The physical link information is stored in the Adabas attributes in Predict for every physical file connected to the database. This information includes the physical file number and the physical Logical Distribution type (how the file is implemented). This type must be either *blank* (simple file) or *E* (expanded).

If the file is expanded, this means that there are several files with the same layout in the same database, and that every file has a different range of ISNs. Adabas Native SQL checks for the physical file with the lowest minimum ISN value (ADALOD LOAD parameter MINISN).

With both simple and expanded files, Adabas Native SQL takes the physical file number from this physical link information. Note that in previous versions of Predict, the physical file number and the logical file number (as exists in the file description) had to be identical. As of Predict Version 3.2 or above, however, the same logical file may have different physical file numbers.

With this kind of distribution, the application is responsible for defining the DBID where every file exists. The AUTODBID-ALL option allows an update program which updates one database and accesses up to five more databases. With AUTODBID-ALL, Adabas Native SQL automatically detects which is the updated database and issues the COMMIT and ROLLBACK commands to it. It also generates different CONNECT and DBCLOSE statements to the different databases.

There is another option AUTODBID-ATM that may be used only in cases that the application will run under the control of the Adabas Transaction Manager (ATM) . With this option Adabas Native SQL does not restrict the number of updated databases within one program. It automatically uses the DBID defined in Predict for every access or update statement while the Commit and Close statements will be pointed to the default database and ATM will take care of the synchronization.

Relational Null Support

Adabas supports relational Null fields. The Null field has an indicator in two binary Byte format which indicates whether the field has a value or is Null. This indicator appears in the Adabas record and value buffers.

The definition of a Null field in Predict is shown by ’R’ or ’U’ in the field Suppression Column.

Adabas Native SQL supports Null fields in the following three clauses:

1. `SELECT` clause

   Every field specified in the SELECT clause which has a Null value indication is generated in the record buffer as two fields. The first field is the Null value indicator as two binary Bytes and its name is the field name, prefixed with "S-". The second field is the field itself.

   This definition is generated for every Null field even if it belongs to a group, or even if `SELECT*` is used.

   When the record is read from the database, a value of zero in the Null field indicator means that the value in the field itself is a real value. A value of "-1" ("x’FFFF’") in the Null field indicator means that the field has no value and is a real Null.

2. `UPDATE/STORE` clauses

   There is a new reserved word "NULL" which may be specified as a value for Null fields. For example:

   ```
   SET field=NULL
   ```

   Adabas Native SQL will move "-1" ("x’FFFF’") to the Null field indicator of the specified field in the record buffer used for updating the file.

   If the user uses the SET clause and specifies a real value or a variable for a field which has a Null value indicator, Adabas Native SQL will automatically reset the Null field indicator of that field. If the user does not specify the SET clause, but initiates the fields in the record buffer by himself, he should also reset or turn on the Null field indicator.

3. `WHERE` clause

   There is an extension to the syntax:

   ```
   WHERE descriptor IS [NOT] NULL
   ```

   This may be used in order to search for all records where the specified descriptor is Null or not Null. This extension is allowed only for descriptors which are defined with the new relational Null support.

Long Alpha field Support

Adabas has a field format "LA", standing for Long Alpha field.

This format represents a variable field whose length may be up to 16K Bytes.

Because it is a variable field, Adabas returns its value together with two binary Bytes in front of the value which represents the actual length of the field (the length includes the two binary Bytes.).

The definition of a Long Alpha field in Predict uses the format "AV".

Adabas Native SQL generates a Long Alpha field as two separate elements in the record buffer. The first element is the field length as two binary Bytes with the name suffixed with "-LEN". Immediately after is the the second element, which is the definition of the field itself as a character string with a total length taken from Predict with the name suffixed with "-TXT".

Because Adabas returns the value of the field in a variable way, it is impossible to have a definition of a field following the Long Alpha field in the record buffer.

For this reason the following restrictions hold:

- the Long Alpha field may be generated only as the last element in the record buffer.
- Only an elementary field is supported as a Long Alpha field (no MU or PE allowed).