Das kursive sollten immer Links werden und selber nicht-kursiv da stehen

ORDER INTERFACE 7.3

Realizing SAP DB requires the exchange of messages between an application and SAP DB tasks. These messages are database requests and the corresponding results.

A log for the interactions between an application process and a SAP DB task is defined in this publication--i.e., defining message formats, message types and message sequences.

Introduction

For the communication between an application process and the corresponding SAP DB task there is the strict rule: the application process has to wait for the answer of the SAP DB task before the next request may be send to this SAP DB task.
An application process may communicate with more than one SAP DB task, but each SAP DB task may have not more than one request at any time.

For communication between an application process and a SAP DB task a message block is used.
The maximum length of this message block must be specified during configuration of each database. The configuration parameter_PACKET_SIZE defines the maximum length of the message block. Usually an application process will provide as much space for the message block as specified with this maximum length. Shorter message blocks are possible if they consist of at least 16 KB.

To avoid copying of the requests and to give the application process the chance to reuse part of one request to build the next one, the request in the message block is not overwritten with the answer send by the SAP DB task. This answer is written behind the request into the same message block. Therefore not the whole message block can be filled with the request. Per default the number of bytes resevered as minimum space for the answer is set to 4 KB. If the request is shorter than _PACKET_SIZE – 4 KB minus some overhead, then the answer may be longer.

The term message is used in this document to describe the requests or the answers send to or from the SAP DB task. When returning the answer, the message block consists of two messages. The starting position of the return message will be different for each request and is returned by the communication routine.

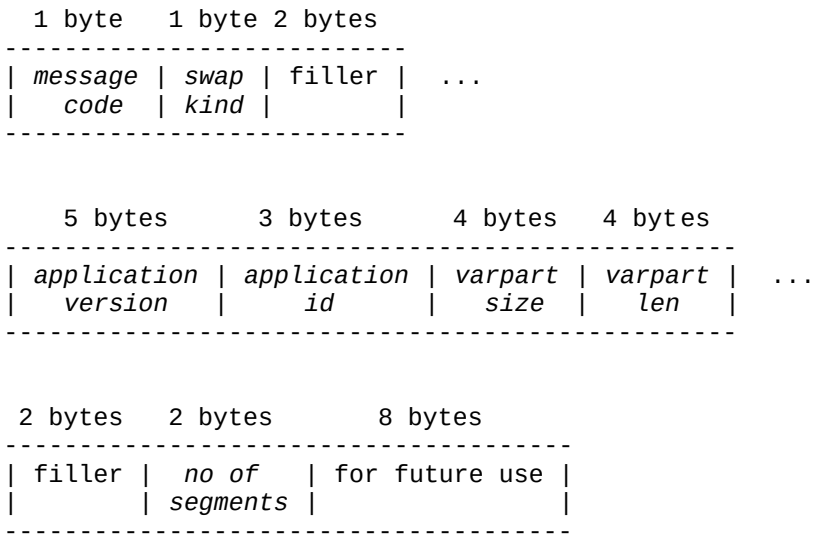
MESSAGE FORMAT

The message format used for the communication between application processes and SAP DB tasks consists of a fixed part, a *message header*, and a *variable part.*

MESSAGE HEADER

The same header is used for both directions; i.e., from application process to SAP DB task and vice versa.

The message header (32 bytes) looks like this :



MESSAGE CODE

The field MESSAGE CODE distinguishes various types of code in the application process, shown here as decimal values:

00 = ASCII
01 = EBCDIC
02 = unused
03 <= 18
used if a termchar set is used for this session; see explanation of the *CONNECT* statement
19 = swapped UCS2
20 = UCS2

A specification in this field is used for interpreting the requests, as well as for returning database contents. The application process sends and receives every information in its own code set which will be internally converted by the SAP DB task according to its needs.

The MESSAGE CODEs 19 and 20 can only be used if the database has the configuration parameter _UNICODE (*see* …) set to YES.

SWAP KIND

The representation of integer values (byte swapping) is declared in the field SWAP KIND.

```
01 = no swapping
    (int2: hi-lo,
     int4: hi4-hi3-lo2-lo1)
02 = full swapping
    (int2: lo-hi,
     int4: lo1-lo2-hi3-hi4)
```

The SAP DB task and not the application process will be responsible for the swapping if the swap kind of the application process and that of the SAP DB task differ from each other. The integers in the requests must be given and the results will be returned with the swap kind of the application process.

APPLICATION VERSION and APPLICATION ID

APPLICATION VERSION

The version of the application must be given, e.g., '70301'.

With this the SAP DB task is able to return the order interface as the application process expects it even if minor changes were defined for the order interface between two versions. Sometimes older application versions have to work with a newer version of SAP DB tasks. For their communication it is necessary.

APPLICATION ID

A short name of the component calling a SAP DB task must be given.

This is used to have some checks for SAP DB components.

VARPART SIZE and VARPARTLEN

VARPART SIZE

The variable part has a variable length which depends on the configuration of the database the message block will be send to and it depends on the application process (see *Introduction*).
.
The length of the existing variable part of the message block must be given in VARPART SIZE. This is NOT the used length of the variable part of the message block. The used length of the variable part is given in VARPARTLEN.

VARPART LEN

VARPART LEN specifies the filled no of bytes in the variable part of the message.

NO OF SEGMENTS

Commands are written to the variable part of the message block. A segment represents either a request to the SAP DB task or its result. NO OF SEGMENTS gives the number of segments, i.e. requests or results.

If one message send to a SAP DB task consists of n segments, the corresponding message send back to the application process will consist of the same number n of segments.

In the message block all requests are one after the other, all results are one after the other. The first result belongs to the first request, the second result to the second request and so on. The first result is written behind the last request.

Several independant segments can be specified in one message. If for building the input one command (one segment) needs the output of another command (segment), these both commands cannot be together in one message.

The segments will be handled one after the other in the given sequence. All subsequent segments will be handled no matter of errors found in the segments handled before. No control flow is available for handling segments.

VARIABLE PART OF A MESSAGE BLOCK

The variable part with its length of VARPART SIZE is (partly) filled with segments, i.e. commands.

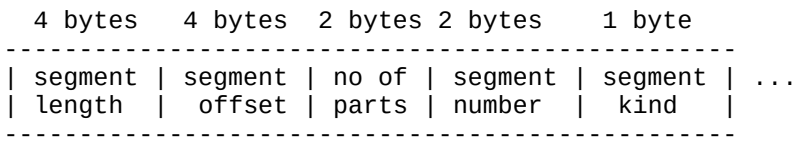
Each segment consists of one ***segment header*** and 0-n different ***parts***. Parts are those components of a segment in which the info concerning this request or result is given, for example the text of the SQL command or the data.

SEGMENT HEADER

Two different types of segment header exist: one for requests from an application process to a SAP DB task and one for the results. Both have the same length, i.e. 40 bytes.
Both of them look alike in the first 13 bytes and differ in the last 27 bytes.

SEGMENT HEADER, common part

The first 13 bytes of the segment header look like this:



SEGMENT LENGTH

SEGMENT LENGTH specifies the length of the segment, i.e the number of used bytes. SEGMENT LENGTH MOD 8 = 0 must be true.

SEGMENT OFFSET

This segment starts at the (SEGMENT OFFSET + 1)-th byte in the variable part.

NO OF PARTS

NO OF PARTS specifies the number of parts belonging to this segment, i.e. to this request or result.

SEGMENT NUMBER

SEGMENT NUMBER specifies the number of the current segment, the first segment has number 1. Segments given in a result use the same numbers starting with 1.

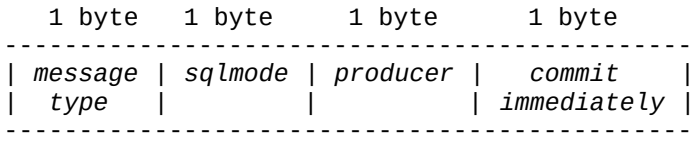
SEGMENT KIND

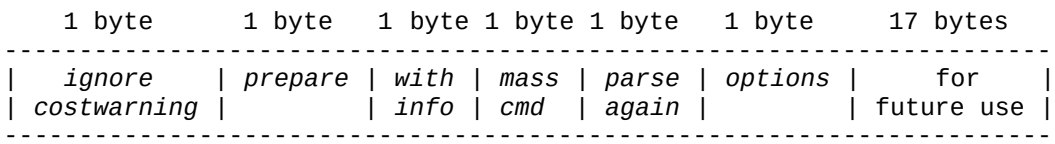
SEGMENT KIND specifies whether a request to the SAP DB task (value 01) or a result (value 02) is involved.

01 = request
02 = result

REQUEST SEGMENT HEADER

In the case of a segment belonging to a request, the 13 bytes specified in *SEGMENT HEADER, common part*, are followed by 27 bytes that look like this:





MESSAGE TYPE

MESSAGE TYPE tells the SAP DB task what kind of request is in the message. The MESSAGE TYPE is shown here as decimal value:

```
01 = unused
02 = DBS
03 = PARSE

05 = SYNTAX

13 = EXECUTE

15 = PUTVAL
16 = GETVAL
17 = LOAD
18 = UNLOAD

25 = HELLO

27 = UTILITY
28 = INCOPY

30 = OUTCOPY
31 = DIAG_OUTCOPY

39 = SWITCH
40 = SWITCHLIMIT
41 = BUFLENGTH
42 = MINBUF
43 = MAXBUF
44 = STATE_UTILITY

50 = WAIT_FOR_EVENT
```

All other numbers are for future use.

The meanings of the MESSAGE TYPEs 02 – 05, 13 – 16 and 25 are explained in the following chapters. The other MESSAGE TYPEs are used internally, e.g. for utilities like backup and recovery. They are not described in this manual.

SQLMODE

The database system can execute correct database applications and applications that are written in accordance with one of the following definitions:

INTERNAL (internal database definition)
ANSI Standard (ANSI X3.135-1992, Entry SQL)
Definition DB2 Version 4
Definition ORACLE7

The database system is able to check whether new applications comply with one of the above definitions. This means, in particular, that any extension above and beyond the selected definition is considered incorrect. Support for DDL statements in other SQL modes, however, is limited.
When connecting to the database system, one of the above-mentioned definitions or the INTERNAL SQL mode can be selected.

```
01 = SESSION_SQLMODE
02 = SAP DB / INTERNAL
03 = ANSI
04 = DB2
05 = ORACLE
```

If nothing special is specified during the CONNECT statement, the SAP DB task will check the given SQL statements against the SAP DB Reference Manual, which describes the functionality of SAP DB in its own SQLMODE (SQLMODE SAP DB). If a user or an application wants to use another SQLMODE, this should be specified during the CONNECT with the option SQLMODE followed by one of the terms ANSI, DB2, ORACLE or (superfluous) SAP DB. The commands will then not be checked against the usual SAP DB syntax but against the syntax as it is the SQL standard or implemented by the systems DB2 or ORACLE.

For some reasons, sometimes users need another SQLMODE for a single statement in a program. Then the SQLMODE for this single statement can be changed using the field SQLMODE. Such a change can be done for the MESSAGE TYPEs DBS, PARSE and SYNTAX, not for the MESSAGE TYPE EXECUTE. The statements will be executed using the SQLMODE specified during the corresponding PARSE.

Usually, SQLMODE will have the value 01 (SESSION_SQLMODE) to indicate that the SQLMODE given during the CONNECT will be used. All the other values change the syntax check for the given statement to the specified SQLMODE.

PRODUCER

PRODUCER must be set to 01 to show that an application process sent this statement. Only SAP DB components are allowed to use the values 02 and 04.

COMMIT IMMEDIATELY

If an application wants to commit (or, if an error occurred, to rollback) immediately after executing the statement, this field must have the value 01. If no immediate commit is wanted, COMMIT IMMEDIATELY must have the value 0.

Independent of the value in COMMIT IMMEDIATELY, a COMMIT or ROLLBACK statement can be specified as the statement to be executed.

This field is useful for applications that usually commit after every SQL statement (Autocommit) and want to avoid additional communication between the application process and the SAP DB task.

IGNORE COSTWARNING

In case, a costwarning was specified for the current user and the execution of a statement would exceed the cost value, the error -4 (COSTWARNING value exceeded) is output.

If the COSTLIMIT values will not be exceeded, the user can force the SAP DB task to suppress the error message and execute the statement by setting IGNORE COSTWARNING to 01 when repeating the statement. Usually, this field should have the value 00.

For future versions of SAP DB it cannot be assured that this value will be considered.

PREPARE

When parsing a SQL statement, the SAP DB task usually does not store the SQL statement, but an internal representation of it, because this will be enough for the execution of the statement.

If the SQL statement must be reparsed, for some reason, it must be given again to the SAP DB task by the application process.

If the SQL statement had been built dynamically, the application process might be unable to know or rebuild the original statement.
In cases in which the application process has no chance to rebuild the statement, PREPARE should be set to 01, thus enabling the SAP DB task to store the SQL statement itself.

As storing the SQL statement needs time and memory, PREPARE should usually be set to 00.
(see also the MESSAGE TYPEs PARSE and EXECUTE)

WITH INFO

Usually, the field WITH INFO should have the value 00.

The value 01 for the field WITH INFO is only allowed together with the MESSAGE TYPE DBS or PARSE.

The field WITH INFO is useful for SQL statements producing an output like SELECT, DECLARE CURSOR, FETCH, RFETCH, and EXPLAIN. Then the SAP DB task will not only return the result of the statement, but also a specification of how to interpret the result. This means, for each output parameter, a description of the name, the data type, the length, and the position where to find the value for this output parameter in the result will be returned. For a full explanation of this description, see the PART KINDs shortinfo and columnnames.

MASS CMD

Sometimes a statement has to be executed for a whole bunch of values, for example insert hundred of rows. It is possible but time consuming to send a message block for each row. It is better to send as much segments in one message block as will fit to handle several rows with one communication. The best way is to send one segment with as many rows as will fit. With this method more rows will be handled with one communication than can be handled if one segment consists of one row. The data-packets will be handled in the given sequence.

Statements like INSERT (no INSERT ... SELECT), UPDATE, DELETE, SELECT (no SELECT DIRECT/FIRST/LAST/NEXT/PREV, which are not available with versions >= 7.4 in any case) and DECLARE can be combined with the field MASS CMD set to 01. Usually, this field will have the value 00.

MASS CMD set to 01 is only allowed together with MESSAGE TYPEs PARSE and EXECUTE, not with DBS.
If MASS CMD is set to 01, several rows can be inserted, updated, deleted or selected by specifying several data-packets and not only one during execution.

In case of a SELECT or DECLARE combined with MASS CMD, all result rows will be in one result table, no matter how many data-packets have been specified. That is different to specifying just one data-packet for the SELECT. If MASS CMD is not set to 01, each SELECT with one data-packet will result in a result table. If the same result tablename is used for these data-packets, each SELECT will overwrite the result table created before and store only its own result rows in this recreated result table. With MASS CMD set to 01, the result rows are appended.

During the PARSE of a statement, a check will be made as to whether the usage of this statement as a mass-command is correct. If this is the case, application info1 in the parse ID will be set to 70 (see the chapter PART KIND parseid) for INSERT, UPDATE and DELETE, and to 114, 115, 116 or 117 for SELECT and DECLARE. Any other value in application info1 indicates that only one data-packet may be given.
If the application process wants to use only one data-packet although the command was parsed with MASS CMD set to value 01, it is allowed to change the application info1 from 70/114/115/116/117 to 0/44/45/46/47. It is not allowed to change the values the other way round.

If MASS CMD was set during PARSE, it is possible, that not all of the data-packets will fit in one EXECUTE message. In this case more than one message can be sent. In the last (even if it it the only one) message in PART ATTRIBUTES in the data part last_packet has to be set. If more than one message will be sent, the first one has to set first_packet, the next ones have to set next_packet and the last one has to set last_packet in PART ATTRIBUTES in its data part.

MASS CMD, error handling

For each segment, only one error number (see *SQLCODE*) can be set. For a segment with several data-packets more than one error may occur. Therefore a special behaviour has to be specified.

If for all data-packets no error is found, SQLCODE is set to 0.

If for one data-packet any error except error 100 ROW NOT FOUND is found, the execution stops at this data-packet. The SQLCODE is set to the found error, *ERRORPOS* indicates the number of the data-packet that caused the error.
The changes done beforehand usually will not be rollbacked. Only if
COMMIT IMMEDIATELY was specified, an implicit rollback will be made.
The subsequent data-packets are not handled in any case and have to be sent again to the SAP DB task if further work shall be done even if an error occurred.

If error 100 occurs with an UPDATE or DELETE statement, usually it is ignored. Only if all data-packets return error 100, SQLCODE is set to 100.

If error 100 occurs with a SELECT (no SELECT … INTO) statement, usually it is ignored. Only if all data-packets return error 100, SQLCODE is handled as usual with SELECTs.

If MASS CMD is set to 01 together with a SELECT … INTO, for each data-packet one result row is returned. For each data-packet returning error 100, a result row filled with ‘FF’ is returned.

PARSE AGAIN

Usually, the field PARSED AGAIN should have the value 00. The value 01 should be used when a statement has to be parsed again after receiving the *SQLCODE* -8 during execution of a statement.

The value 01 will result in locks concerning the catalog information needed for this statement. This assures the correctness of the catalog information used even if another application changes some catalog information of one of the used tables. On the other hand these locks can hide other applications from the chance to work with these tables unless the locks are released by the statement COMMIT or ROLLBACK.

Options

OPTIONS is a set built out up to 8 values. By now only one is used::

bit 0 : no SELECT with implicit FETCH

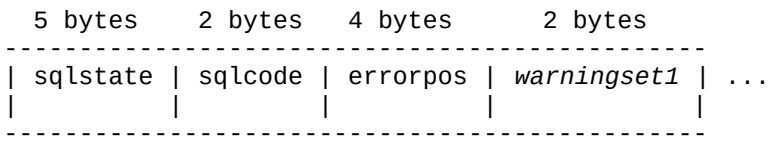
These bits are stored in the byte used for OPTIONS as follows:

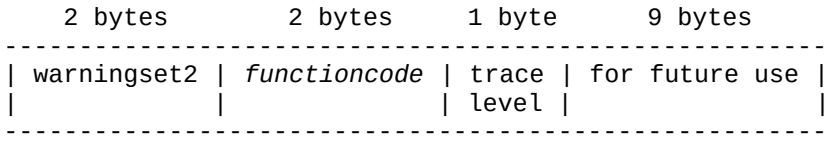
```
--------------------
|  7 6 5 4 3 2 1 0  |
--------------------
```

If a SELECT meets some conditions, the SAP DB task performs a FETCH for the first result rows implicitly. If for some reason the application task does not want this (perhaps because of errors in the code handling this situation), bit 0 can be set. Usually it is 0.

RESULT SEGMENT HEADER

In the case of a segment belonging to a result, the 13 bytes specified above are followed by 27 bytes that look like this:





SQLSTATE

SQLSTATE is returned independent of the SQLMODE. It holds the values that are standardized in SQL2 (ANSI X3.135-1992).

SQLCODE

Each execution of a request sets the SQLCODE. If SQLCODE = 0, then the execution worked correctly. If SQLCODE <> 0, normally a description of the cause of the error is returned. For a description of the possible SQLCODEs, see the manual Messages and Codes.

ERRORPOS

If SQLCODE <> 0, ERRORPOS indicates the position of that part of the SQL statement that caused the error.
If SQLCODE = 0, ERRORPOS is of no use.
For SELECT, DECLARE, INSERT, UPDATE, DELETE with *MASS CMD* set to 01 (see there), ERRORPOS indicates the number of the data-packet that caused the error.

WARNINGSET1

See special chapter.

WARNINGSET2

Unused by now.

FUNCTIONCODE

See special chapter.

TRACELEVEL

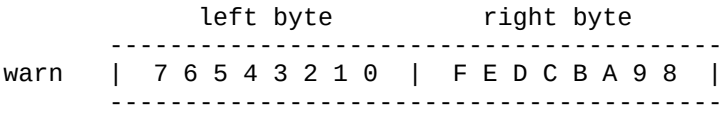
Usually the field TRACELEVEL has the value 00. If for some reason a user wants some tracing for a running application, he can specify the level of tracing wanted for this application within another session. The application to be traced has not to be interrupted. To inform the application about the trace level wanted, its value is given in the field TRACELEVEL.
It depends on the application (precompiler or component) which trace level will result in what kind of tracing.

WARNINGSET1

In some cases independent of the SQLCODE, WARNINGSET1 is not empty having the following meanings:

bit 0 SQLWARN (at least one warning exists)
bit 1 SQLWARN1 (value of a string column was truncated when assigning it to a host variable; NOT set in the SAP DB kernel)
bit 2 SQLWARN2 (NULL values were eliminated from the argument of a function)
bit 3 SQLWARN3 (the number of output columns and the number of host variables is not equal)
bit 4 SQLWARN4 (an UPDATE or DELETE statement does not include a WHERE clause)
bit 5 SQLWARN5 (NOT used)
bit 6 SQLWARN6 (a DATE or TIMESTAMP value was adjusted to correct an invalid date resulting from an arithmetic operation)
bit 7 SQLWARN7 (NOT used)
bit 8 SQLWARN8 (one of two time-consuming functions was used: a sequential search was performed or an ordering was performed while physically building a(n internal) result table)
bit 9 SQLWARN9 (NOT used)
bit A SQLWARNA (NOT used)
bit B SQLWARNB (a TIME value was greater than 99 (or 23 in USA format). The value is corrected modulo 100 (24)).
bit C SQLWARNC (because of the ROWNO specification, no more SELECT result rows are searched, although possibly available)
bit D SQLWARND (in an ordered select statement, INDEX is specified for an optional column)
bit E SQLWARNE (in an ordered select statement INDEX(NAME) is specified and at least one index column of the found row has another value than the specified one)
bit F SQLWARNF (NOT used)

In this field, bit 0 is defined as the right-most bit of the left byte. If it is set, then at least one other bit is set. If it is not set, then there are no warnings.
The warnings are stored in the two bytes of the warningset in the following way:



FUNCTIONCODE

The FUNCTIONCODE tells the application process what kind of SQL statement was executed. The values below 200 refer to the definition in the ORACLE Call Interface V 7.1.
Values 200 and above are only used in the SAP DB system.
Values 1000 and above are used if ORACLE-like array commands (see the field *MASS CMD*) are specified.
Not all of the values below 200 are used by now, but they may be used in future.

```
 1 = create_table
 2 = set_role
 3 = insert
 4 = select
 5 = update
 6 = drop_role
 7 = drop_view
 8 = drop_table
 9 = delete
10 = create_view
11 = drop_user
12 = create_role
13 = create_sequence
14 = alter_sequence
15 = unused1
16 = drop_sequence
17 = create_schema
18 = create_cluster
19 = create_user
20 = create_index
21 = drop_index
22 = drop_cluster
23 = validate_index
24 = create_procedure
25 = alter_procedure
26 = alter_table
27 = explain
28 = grant
29 = revoke
30 = create_synonym
31 = drop_synonym
32 = alter_system_swith_log
33 = set_transaction
34 = pl_sql_execute
35 = lock_table
36 = unused2
37 = rename
38 = comment
39 = audit
40 = noaudit
41 = alter_index
42 = create_external_database
43 = drop_external_database
44 = create_database
45 = alter_database
46 = create_rollback_segment
47 = alter_rollback_segment
48 = drop_rollback_segment
49 = create_tablespace
50 = alter_tablespace
```

```
 51 = drop_tablespace
 52 = alter_session
 53 = alter_user
 54 = commit
 55 = rollback
 56 = savepoint
 57 = create_control_file
 58 = alter_tracing
 59 = create_trigger
 60 = alter_trigger
 61 = drop_trigger
 62 = analyze_table/update_statistics
 63 = analyze_index
 64 = analyze_cluster
 65 = create_profile
 66 = drop_profile
 67 = alter_profile
 68 = drop_procedure
 69 = unused3
 70 = alter_resource_cost
 71 = create_snapshot_log
 72 = alter_snapshot_log
 73 = drop_snapshot_log
 74 = create_snapshot
 75 = alter_snapshot
 76 = drop_snapshot

201 = select_direct
202 = select_first
203 = select_last
204 = select_next
205 = select_prev
206 = fetch_first
207 = fetch_last
208 = fetch_next
209 = fetch_prev
210 = fetch_pos
211 = fetch_same
212 = unused
213 = connect
214 = drop_parseid
215 = close
216 = unused
217 = usage
218 = serverdb
219 = monitor
220 = set
221 = unused
222 = create_domain
223 = drop_domain
224 = describe
225 = alter_password

233 = putval
234 = getval
235 = diagnose
236 = unlock
237 = refresh
238 = clear_snapshot_log
239 = next_stamp
240 = exists_table
241 = commit_release_fc
242 = rollback_release_fc
```

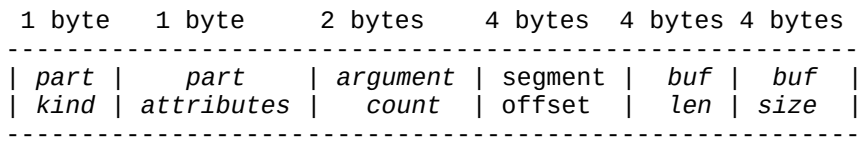
```
 243 = drda_native_ddl_fc
 244 = select_into_fc
 245 = create_database_link_fc
 246 = drop_database_link_fc
 247 = fetch_relative

1003 = insert with MASS CMD set to 01
1004 = select with MASS CMD set to 01
1005 = update with MASS CMD set to 01
1009 = delete with MASS CMD set to 01
1035 = lock_objects with MASS CMD set to 01
1206 = fetch_first with MASS CMD set to 01
1207 = fetch_last with MASS CMD set to 01
1208 = fetch_next with MASS CMD set to 01
1209 = fetch_prev with MASS CMD set to 01
1210 = fetch_pos with MASS CMD set to 01
1211 = fetch_same with MASS CMD set to 01
1244 = select_into with MASS CMD set to 01
1247 = fetch_relative with MASS CMD set to 01
```

VARIABLE PART OF A SEGMENT

Each segment has 0-n parts in its variable part. A part consists of a header and a variable part. In one segment only one part of a special *PART KIND* (see below) will be allowed for a request and be returned.
There is no given sequence for the parts belonging to a segment, and the application process, too, should work with different sequences of parts when checking the result. Only for PART KINDs command and parsid it is necessary to use them as the first part in a request segment.

PART HEADER

A PART HEADER has a length of 16 bytes. It looks as follows:



PART KIND

See special chapter.

PART ATTRIBUTES

See special chapter.

ARGUMENT COUNT

See special chapter.

SEGMENT OFFSET

The segment to which this part belongs starts at the (SEGMENT OFFSET + 1)-th byte in the variable part of the message.

BUF LEN

See special chapter.

BUF SIZE

See special chapter.

PART KIND

The PART KIND tells the SAP DB task and the application process what kind of information follows directly behind the PART HEADER. A segment, i.e. a command, consists of different pieces of information, i.e. of several parts; therefore the PART KIND is the only means to separate these parts from each other.

Each segment may not contain more than one part of one PART KIND.

The PART KINDs are shown here with decimal values:

```
01 = appl_parameter_description
02 = columnnames
03 = command
04 = conv_tables_returned
05 = data
06 = errortext
07 = (not used any more)
08 = modulname
09 = page
10 = parsid
11 = parsid_of_select
12 = resultcount
13 = resulttablename
14 = shortinfo
15 = user_info_returned
16 = surrogate
17 = bdinfo
18 = longdata
19 = tablename
20 = session_info_returned
21 = output_columns_without_parameter
22 = key
23 = serial
24 = relative_pos
25 = abap_inputstream
26 = abap_outputstream
27 = abap_info
28 = checkpoint_info
```

PART KINDs and MESSAGE TYPEs

This table describes parts of which PART KINDs have to be / may be used together with the most common MESSAGE TYPEs.

MESSAGE TYPE **PART KIND**	***DBS***	***PARSE***	***EXECUTE***
appl_parameter_description	——	a(1)	——
columnnames	s(2)	s(2)	s(3)
command	A	A	——
conv_tables_returned	s(4)	——	——
data(5)	a s	——	a s
errortext(6)	s	s	S
modulname(7)	a	a	——
page(8)	——	——	——
parsid	a(9)	S	A s(3)
parsid_of_select	——	s(10)	s(3,11)
resultcount	a(12) s	——	a s
resulttablename	s	s	a s
shortinfo	s(13)	s	s(3)
user_info_returned	s(14)	——	——
bdinfo	——	——	——
longdata	——	——	s(15)
tablename	s	s	——
session_info_returned	s(14)	——	——
output_columns_without_parameter	——	s(13)	s(3)
relative_pos(16)	s	s	——
abap_inputstream	——	——	——
abap_outputstream	——	——	——
abap_info	a	a	A
checkpoint_info	——	——	——

A	mandatory with request from application process to SAP DB task
a	optional with request from application process to SAP DB task
S	mandatory with result from SAP DB task to application process
s	optional with result from SAP DB task to application process

(1) useful only if parameter are used in the command
(2) if output columns are to be described (see *WITH INFO*); given together with *shortinfo* or *output_columns_without_parameter*
(3) in case of SQLCODE = -9
(4) only with SQL statement CONNECT this will be returned, then it is mandatory
(5) only with special SQL statements, the PART KIND data is used together with a DBS request. If results are to be returned or input-parameter-values send to the SAP DB task, , the PART KIND data is used.
(6) in case an error was found

(7) useful only with SQLMODE Oracle
(8) for SAP DB internal use only
(9) for DROP PARSEID
(10) in case of a SQL statement FETCH or CLOSE
(11) for CLOSE
(12) for FETCH with MASS CMD set to 01 / RFETCH
(13) if output columns are to be described (see *WITH INFO*); given together with a part of PART KIND columnnames
(14) will only be returned in case of a CONNECT statement
(15) for INSERT or UPDATE if LONG values are to be sent
(16) in case a FETCH RELATIVE was specified

01 appl_parameter_description

This PART KIND may exist in a request segment together with the MESSAGE TYPE *PARSE*.

In this part, the application process describes the data types and lengths of all parameters used in the SQL statement. In cases in which the SAP DB task has no knowledge about the input parameter's data type and length (for example SELECT cola, :param, colb) or not enough knowledge about its length (SUBSTR (:param, 12)), this information given by the application process will help the SAP DB task. In the first case, it is the only chance to make the SQL statement work; in the second case, it will help to save space in internal structures, because the SAP DB task will not expect a parameter value of maximum possible character length but one with the maximum length of the host variable in the application.

Although this information is only useful for input parameters, it must be given for all parameters in the sequence the parameters are used in the SQL statement.

The description looks like this:

```
         4 bytes                4 bytes
    ----------------------------------------
    | description_1 | ... | description_N |
    ----------------------------------------

DESCRIPTION has the following structure:

     1 byte 1 byte 2 bytes
    ------------------------
    | type | frac | length |
    ------------------------
```

For a description of type, length and frac, see PART KIND *shortinfo*.

Length must be filled with the length of the host variable. If the host variable is longer than 32767 bytes, length should be set to 32767.
If the length of the host variable is not known (then it is a pointer), length must be filled with 0.

The field *ARGUMENT COUNT* in the corresponding part header must have the value N to indicate the number of given descriptions.

02 columnnames

If in the request the SEGMENT HEADER field *WITH INFO* was set to 01 and the MESSAGE TYPE was *DBS* or *PARSE* , then a part of PART KIND columnnames will be returned together with a part of PART KIND *shortinfo* or *output_columns_without_parameter* respectively. Both parts together describe all output parameters of the given SQL statement.

For each output parameter (even if it is not written explicitly), the length (1 byte) of the name of the output column (not of the name of the output parameter) corresponding to the output parameter and then the name itself (using length bytes) are given. Trailing blanks are truncated. Therefore the length of this part is not known beforehand.

03 command

In requests with the MESSAGE TYPEs *DBS*, *PARSE* and *SYNTAX*, this part will contain the SQL statement.

04 conv_tables_returned

If the request was a CONNECT statement, one part of this PART KIND will be returned.

This part consists of 4 (or if a termchar set is used 6) arrays of 256 bytes each. Each of these arrays is a conversion table. The arrays have the following meanings:

```
FROM UPPER ASCII TO SMALL ASCII     BYTE(256)
FROM SMALL ASCII TO UPPER ASCII     BYTE(256)

FROM ASCII TO EBCDIC                BYTE(256)
FROM EBCDIC TO ASCII                BYTE(256)
```

If a termchar set name was specified (the *MESSAGE CODE* for this session will be >= 03 and <= 18), then:

```
FROM TERMCHARSET TO STANDARD ASCII    (BYTE 256)
FROM STANDARD ASCII to TERMCHARSET    (BYTE 256)
```

ARGUMENT COUNT will have the value 4 or 6, thus indicating how many conversion tables belong to this part.

If, for any reason, the application process itself has to convert data, then the application process should use these conversion tables to ensure that the same conversion is done as in the SAP DB task (this is not needed for the interaction with the SAP DB task, because this will do the correct conversions).

05 data

This PART KIND is used for results and, in two cases, for requests.

If the MESSAGE TYPE is *DBS* or *EXECUTE* and a statement returns exactly one result row (SELECT ... INTO, FETCH ... INTO) or some of them (if *MASS CMD* is set to 01), this result will be in a part of PART KIND data. In case of a SELECT whose first results are returned during the execution of the SELECT, these results will be in a part of PART KIND data, too.

If an SQL statement given with the MESSAGE TYPE *PARSE* includes parameters, the corresponding data must be specified during the corresponding *EXECUTE* in a part of PART KIND data. The data must be given as described in the part of PART KIND *shortinfo* returned with the PARSE.

If the SQL statements ALTER PASSWORD, CONNECT, CREATE USER, SET LANGUAGE, SET FORMAT, USAGE ... are specified and a SQL statement given with the MESSAGE TYPE *DBS* contains parameters, the corresponding data must be given in a part of PART KIND data in the same segment. The format of the data is described in this manual together with the corresponding statements.

06 errortext

In the case of SQLCODE <> 0, a part of PART KIND errortext will be returned which consists of up to 256 bytes, describing the error and (in some cases) giving the name of the database object causing the problem.

08 modulname

If the MESSAGE TYPEs are *DBS* and *PARSE*, the application process can specify the name of the module, i.e. of that part of the application that causes the SQL statement.

By now, this field is only used in SQLMODE ORACLE. As result table names are module-local in ORACLE, the SAP DB task uses this information to create, find and destroy the result table used by the module in order to prevent result tables with same names and different source modules from being mixed.

Every statement should specify the modulname because even a DELETE or UPDATE (WHERE CURRENT OF <result table name>) or an INSERT..SELECT FROM <result table name> uses result table names and must therefore be able to distinguish result tables having the same name from each other. The modulname given in other SQLMODEs will have no effect. If no modulname is given, the default-modulname, consisting of blanks, will be used.

09 page

A page is a buffer of 8KB. The layout of pages differs according to their page kinds. This field is not described here, because this PART KIND is for SAP DB use only.

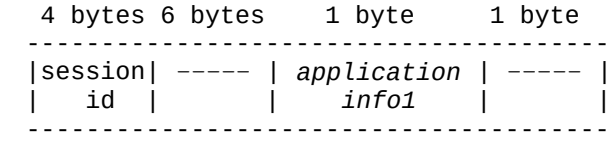
10 parsid

Every request with the MESSAGE TYPE *PARSE* and an SQLCODE = 0 returns a part of PART KIND parsid. The parse ID has a length of 12 bytes.

In a subsequent execution step (see: *EXECUTE*), the unchanged parse ID respectively the parse ID with a change in *application info1* can be provided again, together with a data part belonging to the statement. This causes the execution of the statement.

If the SQL statement *DROP PARSEID* (see there) is specified and the SQL statement is given with the MESSAGE TYPE DBS, the corresponding parsid must be given in a part of PART KIND parsid in the same segment.

The parse ID has the following structure:



——— means: of no interest for the application process

SESSION ID

SESSION ID is the identification of that session in which the corresponding SQL statement was parsed. If several sessions are handled within one application process (because of implicit or explicit reconnect) only those parsids should be used whose SESSION ID is the one of the current session as given as the result of the corresponding CONNECT statement (see PART KIND *session_info_returned*).
All other parsids are not known any more by the SAP DB task. Therefore they cannot be used together WITH MESSAGE TYPE *EXECUTE* and no *DROP PARSEID* should be specified for those unknown parsids.

APPLICATION INFO1

See special chapter

Although all SAP DB statements are parsable, some special statements cannot be executed without being parsed.

If a statement was successfully parsed (SQLCODE = 0), the parse Id consists of binary zeroes with the APPLICATION INFO1 set to 1. If there is no PARAMETER_INFO, this means that the statement was completely executed during the parsing, so that no further EXECUTE can be issued at the time of execution.

APPLICATION INFO1

```
  0 : none
  1 : statement_executed
  2 : (not used any more)
 10 : release_found
 20 : (not used any more)
 30 : not_allowed_for_program
 40 : close_found
 41 : describe_found
 42 : fetch_found
 43 : fetch with MASS CMD found
 44 : mass_select_found
 45 : select_for_update_found
 46 : reuse_select_found
 47 : reuse_select_for_update_found
 60 : dialog_call
 70 : mass_statement
114 : select with MASS CMD found
115 : for_update_select with MASS CMD found
116 : reuse_select with MASS CMD found
117 : reuse_select_for_update with MASS CMD found
```

1 APPLICATION INFO1 is set to 1 in case the statement was executed although the MESSAGE TYPE was PARSE only. This is done with some special diagnose statements and with DDL.

10 APPLICATION INFO1 is set to 10 when COMMIT [WORK] RELEASE or ROLLBACK [WORK] RELEASE was found. The SAP DB task will then close the communication between the application process and the SAP DB task after executing the statement. Same info is given with FUNCTIONCODE

30 APPLICATION INFO1 is set to 30 when a statement was given to the SAP DB task that is known but for internal use only. Such a statement is not allowed in a program.

40..43 The values 40-43 mean that the specified statement (CLOSE, DESCRIBE, FETCH or FETCH with *MASS CMD* (or RFETCH)) was found.
This information can also be found in the *FUNCTIONCODE.*

44 Value 44 means that a DECLARE CURSOR or SELECT but no SELECT DIRECT or ordered SELECT was found.

45 Value 45 means that the statement was a DECLARE CURSOR or SELECT with FOR UPDATE.

46 Value 46 means that the statement was a DECLARE CURSOR or SELECT for which the result set will be physically build.

47 Value 47 means that the statement was a DECLARE CURSOR or SELECT with FOR UPDATE for which the result set will be physically build.

60 The value 60 means that the parse ID belongs to a DB-Procedure.

70 APPLICATION INFO1 is set to 70 when an INSERT, UPDATE or DELETE with *MASS CMD* set to 01 was found (see there).
This information can also be found in the *FUNCTIONCODE.*

114.. APPLICATION INFO1 114 – 117 is the sum of 70 and 44 - 47; i.e., a SELECT or DECLARE (specified with 44 to 47) with *MASS CMD* set to 01 (see there) was given.

11 parsid_of_select

When a FETCH is parsed or a CLOSE is executed, an additional part of PART KIND parsid_of_select will be returned.

This part contains the parse Id of the SELECT statement to which the FETCH / CLOSE statement is connected. This part allows the application process to know which parse Ids belong together or which result table was closed.

The parsid_of_select has the same structure as the 10 parsid. Only if the SELECT was not parsed, the 12 bytes are filled with binary zeroes.

12 resultcount

Every INSERT, UPDATE, DELETE, SELECT, DECLARE CURSOR, FETCH, RFETCH statement will result in at least one part of PART KIND resultcount. This is true for all MESSAGE TYPEs except for SYNTAX. The resultcount specifies the number of rows inserted, updated, deleted, found or fetched.

In the cases in which the number of rows is unknown, but at least 1 (for example, if a result table is not physically build), resultcount has the value -1.

The part contains one UNDEFSIGNAL and a number of the format fixed (10,0). For a description of how this number is written into the part, see the chapter *DATA FORMAT*.

If an SQL statement like FETCH with *MASS CMD* set to 01 or *RFETCH* is used, the number of wanted result rows is specified by the application process in one part of PART KIND resultcount.

During the execution of INSERT, UPDATE, DELETE, SELECT, and DECLARE (all with MASS CMD set to 01), the application process must specify, how many rows had been previously handled for this ORACLE-like array command. The SAP DB task does not remember this value which is returned to the application process during the previous execution. This value must be a part of PART KIND resultcount.
This given number plus the number of rows handled during this execution will be returned in a part of PART KIND resultcount. This will happen even in case of an error.

13 resulttablename

For some reasons, the application process often needs to know the name of the result table for which the parsing or execution was done. To avoid that the SQL statement is checked twice (once by the application process, once by the SAP DB task; the two checks may differ from each other), the SAP DB task returns the result table name when the *FUNCTIONCODE* is select, select with *MASS CMD* set to 01, explain, select_direct, select_first, select_last, select_next, or select_prev.

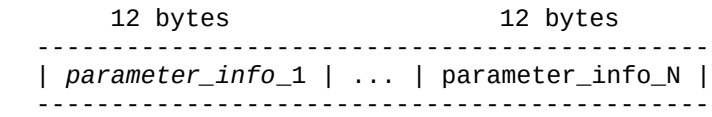
If the application process uses the MESSAGE TYPE PARSE or SYNTAX, a parameter can be used instead of a result table name. Then the result table name must be specified when the statement is being executed (see the MESSAGE TYPEs PARSE and EXECUTE). The result table name must be specified in a part of PART KIND resulttablename of its own.
In both cases, trailing blanks are truncated. Therefore, there will be an empty part of PART KIND resulttablename, because all blanks were truncated when an unnamed result table had been specified.
If the SQL statement was a single select or SELECT DIRECT/FIRST/LAST/NEXT/PREV, no result table exists. To show this, the result table name is filled with binary zeroes.

14 shortinfo

If the request has the MESSAGE TYPE *PARSE*, there will be a part of PART KIND shortinfo when returning. This part contains a description of all input and output parameters used in this statement.
Even if there is no parameter specified in the SQL statement, such a part will exist, consisting of the PART HEADER with ARGUMENT COUNT = 0.

If the request has the MESSAGE TYPE *DBS* and the field *WITH INFO* has the value 01, a part of PART KIND shortinfo will be returned together with a part of PART KIND columnnames describing all the output parameters (even if these parameters are not written explicitly). In this case at least one parameter will be described).

For the description of one parameter, 12 bytes are needed. The whole description looks like this:



These PARAMETER_INFOs apply for all input and output parameters in the order of their occurrence. This means that a record column is described several times, if necessary.

PARAMETER_INFO

PARAMETER_INFO has the following structure:

```
1 byte 1 byte  1 byte 1 byte 2 bytes 2 bytes  4 bytes
-------------------------------------------------------
|mode| in/out | data | frac |length | in/out | bufpos |
|    |  type  | type |      |       | length |        |
-------------------------------------------------------
```

MODE

MODE indicates whether or not NULL values are accepted as input or output parameters for this column, whether a default value was specified for this column, or whether this is the input parameter used as escape character.

MODE is a set built out of the following 4 values:

bit 0 : NULL is not accepted
bit 1 : NULL is accepted
bit 2 : column has the default
bit 3 : input parameter for escape character

These bits are stored in the byte used for MODE as follows:

```
---------------------
|  7 6 5 4 3 2 1 0  |
---------------------
```

IN/OUT TYPE

IN/OUT TYPE specifies whether the parameter involved is an input or output parameter or a parameter used for input and output.

```
00 : Input
01 : Output
02 : In- and Output; only used in DB-Procedures
```

DATA TYPE

see special chapter

LENGTH and FRAC

LENGTH and FRAC contain the length and, for fixed number columns, the number of decimal places. In all other cases, FRAC will have the value 0.

IN/OUT LENGTH

see special chapter

BUFPOS

BUFPOS defines the position where--upon the following *EXECUTE* statement--either the UNDEFSIGNAL byte of the corresponding column must start in the data part or where the output result will be.

DATA TYPE

```
00 : Fixed number
01 : Float number
02 : Char ASCII
03 : Char EBCDIC
04 : Char BYTE / RAW
05 : future use
06 : LONG ASCII
07 : LONG EBCDIC
08 : LONG BYTE / LONG RAW
09 : unused
10 : DATE
11 : TIME
12 : Float number from virtual column
13 : TIMESTAMP
14 : internal use
15 : internal use
16 : internal use
17 : internal use
18 : unused
19 : LONGFILE ASCII
20 : LONGFILE EBCDIC
21 : LONGFILE BYTE
22 : unused
23 : Boolean
24 : Unicode
25 : future use
26 : future use
27 : future use
28 : future use
29 : Smallint
30 : Integer
31 : Varchar ASCII
32 : Varchar EBCDIC
33 : Varchar Byte / Varchar RAW
34 : LONG Unicode
35 : LONGFILE Unicode
36 : Varchar Unicode
37 : future use
38 : internal use
```

DATA TYPE specifies the data type that was given during a CREATE TABLE or ALTER TABLE or that was the result of an expression.

Usually, character columns will have the code attribute ASCII; i.e., the data type Char ASCII or Varchar ASCII. If another code attribute has been explicitly specified, this can be changed to EBCDIC, BYTE or UNICODE.

In tables created in SQLMODE DB2, character columns will have the code attribute EBCDIC to ensure the same ordering sequence of values in the table as in DB2.
The use of arithmetic in expressions (virtual columns) will produce FLOAT values very soon. These values are signalled by the special DATA TYPE = 12, which gives the interactive components the opportunity to still output them as FIXED values.

With the data types 06 – 08, 19 – 21 and 34 - 35 (LONG/LONGFILE), there is no description of the long column itself. In the PARAMETER_INFO, there is a description of a block in

which the current position, length and so on of the long column are written. The long column can be found according to the position given in the block, not according to BUFPOS. A description of how this block looks like can be found in the chapter DATA FORMAT.

IN/OUT LENGTH

IN/OUT LENGTH specifies either how long this value must be or how long it will become (if necessary, by filling it up).

The filling will be done with blanks for values of types Char ASCII/EBCDIC, Unicode, Varchar ASCII/EBCDIC/Unicode, DATE, TIME, and TIMESTAMP.
The filling has to use hexadecimal 00 for all kinds of numbers, Char BYTE, Varchar BYTE, and Boolean.

With the types Char ASCII, Char EBCDIC, Char BYTE, Varchar ASCII, Varchar EBCDIC, Varchar BYTE and Boolean, the IN/OUT-LENGTH will be = LENGTH+1, because the UNDEFSIGNAL byte increases the length by one. For a description of UNDEFSIGNAL, see the chapter DATA FORMAT.

With the types Char UNICODE and Varchar UNICODE the IN/OUT-LENGTH will be = (2 * LENGTH) + 1, because each character needs two bytes for representation and the UNDEFSIGNAL byte increases the length by one. For a description of UNDEFSIGNAL, see the chapter DATA FORMAT.

For numeric values, the IN/OUT LENGTH can be calculated from LENGTH by means of ((LENGTH + 1) DIV 2) + 2. This does not apply for the values output by the SQL functions SUM and AVG. Here, the result implicitly provided is always of the FLOAT(38) type. By using the FIXED function, the user can provide an implicitly created FLOAT(38) result column as a FIXED column with the appropriate LENGTH and FRAC specifications.

15 user_info_returned

With CONNECT statements, one part of PART KIND user_info_returned is returned. This part contains of several information, each with one byte describing the length of the information and followed by the information itself with the given number of bytes.
The information will be given in this sequence :
the SYSDBA of the SERVERDB to which the current user is connected (max 64 bytes),
the SYSDBA of the SERVERDB where the current user was created (max 64 bytes),
the user type (RESOURCE, STANDARD, DBA, OPERATOR) of the current user (max 16 bytes),
the name of the usergroup to which the current user belongs (max 64 bytes).

Usually, the lengths of the names of user and usergroups will not exceed 32 bytes. Only in the case of MESSAGE CODE UCS2 or swapped UCS2, the maximum length will be 64.

No UNDEFSIGNAL is placed in front of each piece of information.

16 surrogate

This PART KIND is for SAP DB use only. It is used together with fast load, i.e. page-wise inserting of rows. This PART KIND tells the application/the SAP DB kernel how far the surrogates are used for this table.

17 bdinfo

This PART KIND is for SAP DB use only. It exists together with the PART KIND page and contains a description of how to interpret the pages in that part.

18 longdata

Any data given in a request or result with the MESSAGE TYPE *PUTVAL* or *GETVAL* is included in one part of PART KIND longdata. This PART KIND is used in the result segment, too, in case an INSERT or UPDATE was executed and the description blocks for LONG columns which have to be filled using MESSAGE TYPE PUTVAL are sent to the application process. For a description, compare the two MESSAGE TYPEs PUTVAL and GETVAL..

19 tablename

If the SQL statement SELECT or DECLARE CURSOR is given and only one tablename is specified in the from-clause, then the username (if specified) and the tablename are returned.

This part is returned in the form ["owner".]"tablename" whereas each name is specified with its current length, not with the maximum length of a name. At least one SAP DB component needs these names to be able to build up a statement like
UPDATE/DELETE ... WHERE CURRENT OF ...

20 session_info_returned

With CONNECT statements, one part of PART KIND session_info_returned is returned. It is returned for statements like SET NLS…, too.

This part contains 2208 bytes.

BYTE	LEN	VALUE
1	1	UNICODE-databaseYES=1/NO=0
2	4	session_id as used in the first 4 bytes of the pars Id
6	1	NLS_DATE_FORMAT length of clear format
7	100	NLS_DATE_FORMAT clear format
107	1	NLS_DATE_FORMAT length of coded format
108	50	NLS_DATE_FORMAT coded format
158	6	NLS_DATE_LANGUAGE clear
164	6	LANGUAGE
170	1	DATE_FORMAT (see tgg_datetimeformat)
171	4	NUMERIC_CHARACTERS
175	2	empty (zero), alignment-filler
177	16	empty (zero)
193	2000	DICTIONARY (without code tables)
2193	5	PAGE SIZE
2198	3	empty (zero), alignment-filler
2201	5	Kernel version
2206	3	empty (zero), alignment-filler

21 output_columns_without_parameter

In special cases of SELECT and DECLARE CURSOR one part with PART KIND output_columns_without_parameter is returned as result of the parsing. This PART KIND describes the data types and lengths and so on of the output columns of the select. Because it is a SQL statement resulting in a result set, no parameters are specified in the SQL statement for these output columns.

If this part is returned, the SAP DB task will return as many result rows as possible (as exist or as fit into the message block) during the execution of the SELECT statement. If all results were returned during the SELECT, no FETCH should be executed. The result table is not known any more and cannot be specified in another from-clause.
If not all result rows fitted into the message block, the first n ones are returned. The last ones have to be fetched as usual. In this case the result set is known as long as usual.

The description of the output columns looks like those in parts of PART KIND *shortinfo* (see there).

22 key

This PART KIND is for SAP DB use only.

23 serial

This PART KIND is for SAP DB use only.

24 relative_pos

This PART KIND looks like the PART KIND resultcount. It is send back to the application process in case a 'FETCH RELATIVE' was found with the position specified in the command, not given via parameter. This is necessary to tell the application process the current position in the result set. Otherwise the application process had to parse the SQL statement.

25 abap_inputstream and 26 abap_outputstream

These PART KINDs are for SAP DB use only. It is used between a SAP DB task and a precompiler task to send the contents of ABAP-tables.

27 abap_info

This PART KIND consists of two parts: the name of the program/ABAP/testfile the given SQL statement is written in and the linenumber in this program/ABAP/testfile this SQL statement can be found. The linenumber is specified as char value.

Each part consists of 1 byte giving the length of the information following directly behind this byte and the information itself with the length given in the corresponding byte.

28 checkpoint_info

This PART KIND is for SAP use only.

PART ATTRIBUTES

PART ATTRIBUTES is a set built of the following bits:

bit 0 : last_packet
bit 1 : next_packet
bit 2 : first_packet

The bits are stored in the byte used for PART ATTRIBUTES as follows:

```
--------------------
|  7 6 5 4 3 2 1 0  |
--------------------
```

Statements with *MASS CMD* set to 01 are used for ORACLE-like array commands. These will be parsed with the MESSAGE TYPE PARSE and then be executed.

It may happen that not all the data stored in the application process array can be sent to the SAP DB task with one message block, so that it must be divided into several blocks. To show the SAP DB task that the message block is the first one, the last one or one message block in between for this array command, PART ATTRIBUTES have to be set.

Only none or exactly one of those bits defined by now may be set at one time.

Last_packet must be set in the segment in which the last data-packet was given to the SAP DB task, even if this is the only message block for this statement.

In cases of FETCH with MASS CMD set to 01 or the SELECT which will do such FETCH implicitely last_packet is set to show the application process that no more results can be found and that no further FETCH is necessary.

ARGUMENT COUNT

This field specifies the number of arguments in the part.

In the case of 03 (command), 06 (errortext), 08 (modulname), 10 (parsid), 11 (parsid_of_select), 12 (resultcount), 13 (resulttablename), 15 (user_info_returned), 19 (tablename), 20 (session_info_returned), 24 (relative_pos), and 27 (abap_info) this is always 1.

In the case of 05 (data), this value is usually 1.
Only together with SQL statements which were parsed with *MASS CMD* set to 01 when more than one data-packet is given to the SAP DB task, this value can be higher than 1. ARGUMENT COUNT then specifies the number of data-packets.

In the case of a result of FETCH with MASS CMD set to 01 or RFETCH or a SELECT which returns the first n result rows, ARGUMENT COUNT may be higher than 1, too.

In the case of 01 (appl_parameter_description), 02 (columnnames), 14 (shortinfo), and 21 (output_columns_without_parameter), ARGUMENT COUNT specifies the number of parameters described in this part.

In the case of 04 (conv_tables_returned), ARGUMENT COUNT will be 4 or 6, dependent of the specification of a termchar set in the CONNECT statement.

BUF LEN

This field specifies the number of bytes used in the variable part of this part, i.e. the part behind the PART HEADER.

Every part must start at a position in the variable part of the message block, so that 'number-of-bytes-before' MOD 8 = 0 is true. As the PART HEADER has a length of 16 bytes, the application process must only care about the number of bytes between two PART HEADERs or between the PART HEADER and the next SEGMENT HEADER. Nevertheless, BUF LEN must specify the number of used bytes, which usually cannot be divided by 8. Then, between two parts, there will be a 'hole', what is correct.

BUF SIZE

This field specifies how many bytes are left in the variable part of the message block, counting the first byte behind the PART HEADER as the first one.

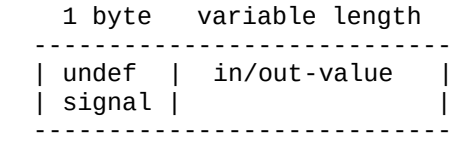
For the first part of the first segment, this must be VARPART SIZE - length-of-a-SEGMENTHEADER - length-of-a-PARTHEADER. For the second part in the first segment, this must be VARPART SIZE - length-of-a-SEGMENTHEADER - length-of-a-PARTHEADER(first part) - BUF LEN-of-first-part - 'hole' - length-of-a-PARTHEADER(second part).

VARIABLE PART OF A PART

According to the PART KIND specified in the PART HEADER, the information must start directly behind the PART HEADER and has a length of 0 to BUF SIZE bytes. The length of the information in the variable part is stored in BUF LEN.

DATA FORMAT

Values of host variables must be and will be specified in the following manner:



UNDEFSIGNAL = 'FF' means that the value of the corresponding column = NULL (undefined). In that case, IN/OUT-VALUE has the proper length for the particular column, but is not relevant. Usually it is filled with hexadecimal 00.

UNDEFSIGNAL = 'FE' can only be found with output. It means that IN/OUT-VALUE has the proper length for the particular column, but is not relevant. This is because there was an arithmetical overflow during the calculation of the corresponding arithmetic expression.

UNDEFSIGNAL = 'FD' is for SAP DB use only. It means that the corresponding column was defined with a default value and that this default value should be used. The given IN/OUT-VALUE is not relevant but has the proper length.

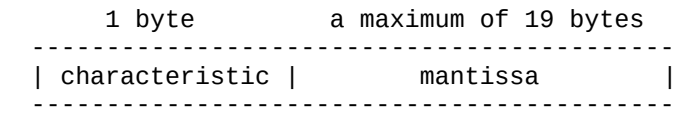
All other values of UNDEFSIGNAL mean that IN/OUT-VALUE contains the relevant value. Usually the UNDEFSIGNAL should have / will have the same value as the byte, the value is filled with, if necessary. The filling will be done with blanks for values of types Char ASCII/EBCDIC, Varchar ASCII/EBCDIC, DATE, TIME, and TIMESTAMP. The filling has to use hexadecimal 00 for all kinds of numbers, Char BYTE, Varchar BYTE, and Boolean. For value of Unicode types the filling has to be done with blanks (2 bytes), but the UNDEFSIGNAL will be hexadecimal 01.

The length of the IN/OUT-VALUE is determined by the maximum number of positions this column can occupy. With character columns, it actually equals this maximum number; with unicode character columns it actually equals twice this maximum number; with number columns, it is the result of the internal length formula ((LENGTH + 1) DIV 2) + 2.

Numberformat

NUMBER (FIXED, FLOAT, SMALLINT and INTEGER)

Within the SAP DB system, number values are stored and processed in special a representation. This has the following structure:



This yields a maximum length of 20 bytes for number representation within SAP DB.

With each number column definition in SAP DB, the internal length of the number representation is defined by the formula:

((n+1) DIV 2) + 1

where n specifies the total number of positions in the number column--that is: the desired precision for this column. This n is specified as part of the number column definition, such as:

fixed (n,m)

or

float (n).

m defines the number of decimal places in a fixed number column.

Since n should not be larger than 38 in SAP DB (in some SQLMODEs larger ones are changed to 38), the formula defines a maximum length of 20 bytes for number representation.

The characteristic of a number Z = z * 10 ** e has one of the following settings:

```
192 + e   if Z > 0
128       if Z = 0
 64 - e   if Z < 0
```

The characteristic 0 is not used.

The following applies for the mantissa z: 0 <= z < 1. The mantissa z is scaled; i.e., displayed without leading zeroes. The mantissa digits are stored in the form of packed decimal numbers (1 digit per half-byte). The mantissa has the following form:
z = scaled digits if Z >= 0
z = scaled digits as the ten's complement if Z < 0.
The ten's complement is formed by subtracting the last digit of the mantissa from 10, the previous digits from 9.
If necessary, the mantissa is filled up with binary zeroes.

This mantissa and exponential form was selected so that simple string comparisons can be performed, when comparing numbers.

Fixed number columns have a maximum absolute exponent of 38.
Float number columns have a maximum absolute exponent of 63.

Character Data

The CHAR column or VARCHAR column with the code type ASCII or EBCDIC, with a maximum length of 8000 bytes, is translated into either STANDARD ASCII or STANDARD EBCDIC, depending on the specified code type of the application process, and then, during output, adapted again to the code type or termchar set of the application process. At the same time, the column--written in left-adjusted fashion--is filled up with blanks, if necessary.

A CHAR column or VARCHAR column with the code type BYTE is filled up with binary zeroes, if necessary. It is not converted even in case application process and SAP DB task uses different code types (ASCII/EBCDIC).

A CHAR column or VARCHAR column with the code type UNICODE may only exist in a database with configuration parameter _UNICODE set to YES. A column with code type UNICODE has a length of up to 4000 character = 8000 bytes.

The input and output of UNICODE data is given as UCS_2 or the swapped form of UCS_2, no matter of the value for MESSAGE CODE.

This column--written in left-adjusted fashion--is filled up with unicode blanks, if necessary.

Date- and Time Data

DATE

The data type DATE has a length of 10 bytes and the filling convention depends on the date-and-time format in use (INTERNAL, ISO, USA, EUROPE, JIS, ANSI). If necessary, the column is filled up with blanks.
The date output also has the representation depending on the date-and-time format in use. The specified MESSAGE CODE controls ASCII/EBCDIC/termchar set conversion, if this should be necessary.

TIME

The data type TIME has a length of 8 bytes and the filling convention depends on the date-and-time format in use (INTERNAL, ISO, USA, EUROPE, JIS, ANSI). If necessary, the column is filled up with blanks.
The time output also has the representation depending on the date-and-time format in use. The specified MESSAGE CODE controls ASCII/EBCDIC/termchar set conversion, if required.

TIMESTAMP

The data type TIMESTAMP has a length of 26 bytes and the filling convention depends on the date-and-time format in use (INTERNAL, ISO, USA, EUROPE, JIS, ANSI or NLS_DATE_FORMAT in case of SQLMODE Oracle). If necessary, the column is filled up with blanks.
The timestamp output also has the representation depending on the date-and-time format in use. The specified MESSAGE CODE controls ASCII/EBCDIC/termchar set conversion, if required.

BOOLEAN Data

The data type BOOLEAN has a length of 1 byte and only the values 00 (false) and 01 (true) are accepted.

LONG Data

Values of data types LONG and LONGFILE (shortly named LONG) have a length of up to 2 GB. It is impossible to read/write such a value with one message. Therefore several reads (see *GETVAL*) or writes (see *PUTVAL*) are performed with one data portion each.

To describe the portion and the value it belongs to, a block of 40 bytes in length (plus one byte for the UNDEFSIGNAL) is written. The *shortinfo* given for a LONG column describes the position and length of this block, not of the LONG value.

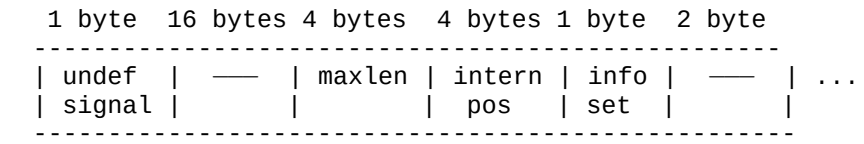
The current portion of the LONG column can use the whole variable part not used so far of the part the data is written to. The maximum length of the variable part is specified with *BUF_SIZE*.

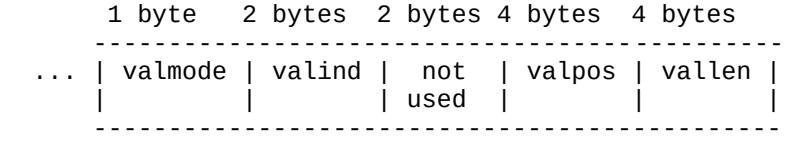
The specified *MESSAGE CODE* controls ASCII/EBCDIC/termchar set/Unicode conversion for LONG, if required.

See PUTVAL/GETVAL for a description of the processing of LONG columns in DML, SELECT, DECLARE CURSOR and FETCH statements.

LONG Description Block

The description block has the following structure:





——— : of no interest for the application process

For a LONG value the value NULL is possible, therefore UNDEFSIGNAL may be hexadecimal FF. Any other value for UNDEFSIGNAL (usually hexadecimal 00) makes the description block a valid one.

MAXLEN

This field will be used for reading LONGs. When the first part of the LONG value is read, the length of the whole LONG is given in MAXLEN to allow the application process respectively the application to assign enough storage space for the whole LONG value.

INTERN POS

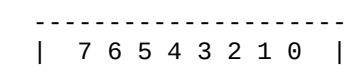
INTERN POS specifies the first position in the column which must be read/written next. It always starts with 1. For writing LONG values the application process has to initialize this value with 1. Afterwards the application process may not change this value. It will be changed by the SAP DB task.

INFO SET

INFO SET is a set consisting of several elements. Nearly all of them are used only by the SAP DB task. Only bit 2 may be used by the application process.

bit 0 : not used by application process
bit 1 : not used by application process
bit 2 : no_close
bit 3 : not used by application process
…
bit 7 : not used by application process

These bits are stored in the byte used for INFO SET as follows:



During reading a LONG column value, the application process specifies the maxmum number of bytes which can be returned for this LONG value. Even if more than this number of bytes is stored in the database, the LONG value is closed, i.e. reading is finished for this LONG value. To prevent the closing because reading will be done in smaller pieces, no_close has to be sent during GETVAL messages. Then the LONG column will not be closed and the next piece of the LONG column can be read with the next GETVAL message.

See vm_close and GETVAL for a full description.

VALMODE

See special chapter

VALIND

VALIND is a field for private use by the application process. In this for example the number of the corresponding parameter or something like this may be stored. The SAP DB task will not change it.

VALPOS

VALPOS gives the position of the data portion of the LONG column in the variable part of the current part of PART KIND data/longdata.

VALLEN

VALLEN specifies the length of the data portion. The data portion has no UNDEFSIGNAL.

If a LONG descriptor has to be initialized for the EXECUTE of an INSERT or UPDATE with LONGs, the first 20 bytes of the block should be set to binary zero. INTERN POS has to be set to 1, INFO SET to 0. The 26^{th} and 27^{th} byte should be set to hexadecimal 00. VALMODE, VALPOS and VALLEN have to be set to values according to the description in PUTVAL, Communication Protocol.

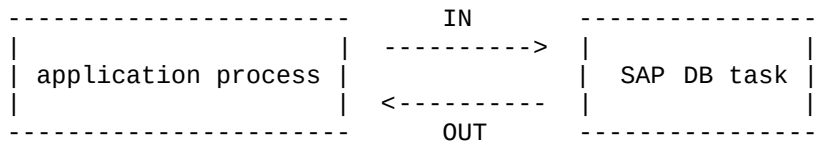
VALMODE

VALMODE specifies how this description block must be interpreted.

00 (vm_datapart) one of several data portions for this LONG column
01 (vm_alldata) the whole LONG column value
02 (vm_lastdata) the last of several data portions for this LONG column
03 (vm_nodata) no data portion for this LONG column in this segment
04 for internal use
05 (vm_last_putval) no description block for a data portion; all data portions for all LONG columns of this row have been sent, the handling of this row is finished
06 (vm_data_trunc) is used in GETVAL results:
the SAP DB task was told (in VALLEN) how much space is left in the host variable to store the returned LONG value part and finds that this will not be enough.
07 (vm_close) the application process wants to close the half-read column after sending at least one GETVAL message with no_close set for this LONG descriptor

MESSAGE TYPES

The descriptions of the MESSAGE TYPEs contain the directional specifications IN and OUT. These specifications refer to the following model of interaction:



DBS

```
*IN  - PART KINDs
            command
            modulname   (optional)
            resultcount (for FETCH with MASS CMD set to 01/RFETCH)
            data        (for special commands)
            parsid      (for DROP PARSEID)
            abap_info   (optional)

*OUT - PART KINDs
            errortext
            resulttablename (for SELECT, DECLARE
                             in case of error 100)
                  OR
            conv_tables_returned    (for CONNECT)
            user_info_returned      (for CONNECT)
            session_info_returned   (for CONNECT)
                  OR
            resultcount     (for DML, SELECT, DECLARE, FETCH..)
            resulttablename (for SELECT, DECLARE)
            tablename       (for SELECT, DECLARE)
            data            (for SELECT .. INTO, FETCH)
            columnnames     (if WITH INFO was specified and for DESCRIBE)
            shortinfo       (if WITH INFO was specified and for DESCRIBE)
            relative_pos    (for FETCH RELATIVE with parameter)
```

All the SAP DB functions except the Utilities are available via this MESSAGE TYPE. The statement string created in the manner described in the SAP DB Reference Manuals is provided in a part of PART KIND command.
The statement is checked for correct syntax, analysed and executed with one communication between application process and SAP DB task. The SAP DB task does not have any memory of finished statements sent with this MESSAGE TYPE except for diagnostic info if this is switched on.

This MESSAGE TYPE allows the specification of a modulname for all statements. This modulname is used to distinguish module-local result table names in SQLMODE ORACLE.

A part of PART KIND data may be only specified together with the SQL statements ALTER PASSWORD, CONNECT, CREATE USER, SET LANGUAGE, SET FORMAT, USAGE ... (see there) if these statements contain parameters.

A part of PART KIND parsid may be only specified together with the SQL statements DROP PARSEID (see there).

PARSE

```
*IN  - PART KINDs
            command
            modulname                   (optional)
            appl_parameter_description (optional)
            abap_info   (optional)

*OUT - PART KINDs
            errortext
                  OR
            shortinfo
            parsid
            parsid_of_select (for FETCH, RFETCH)
            resulttablename  (for SELECT, DECLARE)
            output_columns_without_parameter
                             (for SELECT, DECLARE)
            columnnames      (if WITH INFO was specified)
            relative_pos     (for FETCH RELATIVE with parameter)
```

This MESSAGE TYPE causes the parsing of a SAP DB statement in an application program. This means, the syntactical and semantical (existence, privileges,…) check is done and the statement converted into an internal structure whose identification is a so-called parse ID.

The statement string created in the manner described in the SAP DB Reference Manuals is provided in the part of PART KIND command. As a result of this parsing, SAP DB returns the 12-byte parse ID in the part of PART KIND parsid.

In a subsequent execution step (see: EXECUTE), this unchanged parse ID or parse ID with changed field APPLICATION INFO1 has to be provided again, usually together with a data part belonging to the statement. This will then cause the execution of the statement.

The reason for separating the parsing process from execution is most evident when calling SAP DB from a programming language: Whenever the same statement is called repeatedly (with different data each time), it is not necessary to repeat the check of the statement syntax, user authorization and existence of SAP DB objects and it is not necessary to repeat the conversion of the statement into internal structures as they are needed for execution.

If a statement is faulty, then no part of PART KIND parsid is returned, but one of PART KIND errortext.

SYNTAX

```
*IN  - PART KINDs
            command

*OUT - PART KINDs
            errortext
                  OR
            no part
```

This MESSAGE TYPE causes limited parsing of a SAP DB statement in an application program.

The statement string--created in the manner described in the SAP DB Reference Manual --is provided in the part of PART KIND command. Unlike the MESSAGE TYPE PARSE, SYNTAX does not perform an existence test for the specified table or column names, nor does it convert the statement into internal structures needed for execution. Only the formal statement syntax is tested.

Only with MESSAGE TYPE SYNTAX, runtime parameters (<NNN>) can be specified instead of table and column names.

As a result of the syntax check, the SQLCODE is set and--in the event of a syntax error--ERRORPOS is set to the error position and a part of PART KIND errortext is returned.

You cannot use the MESSAGE TYPE SYNTAX instead of PARSE to separate parsing and execution. To do so, a PARSE must be requested at runtime (once the runtime parameters have been substituted).

EXECUTE

```
*IN  - PART KINDs
            parsid
            data              (optional)
            resultcount       (for RFETCH,
                                   INSERT, UPDATE, FETCH ...
                                   with MASS CMD set to 01)
            resultablename    (if not given in statement
                                   during PARSE)
            abap_info         (optional)


*OUT - PART KINDs
            errortext
            resulttablename   (for SELECT, DECLARE
                              in case of error 100)
            resultcount       (for INSERT, UPDATE, DELETE
                                   with MASS CMD set to 01)
                  OR
            resultcount       (for DML, SELECT, DECLARE, FETCH..)
            resulttablename   (for SELECT, DECLARE)
            data              (for SELECT, DECLARE, FETCH)
            longdata          (for statements with LONG column)
            parsid_of_select  (for CLOSE)

                  OR (in case of SQLCODE = -9):
            shortinfo
            parsid
            parsid_of_select  (for FETCH, RFETCH)
            resulttablename   (for SELECT, DECLARE)
            output_columns_without_parameter
                              (for SELECT, DECLARE)
            columnnames       (if WITH INFO was specified during
                                    parsing)
```

This MESSAGE TYPE executes previously parsed SAP DB statements. To be able to do so, the parse ID obtained in the parse step must be specified in the part of PART KIND parsid.

The data part belonging to the statement is provided in a part of PART KIND data. In case the statement to be executed was parsed with MASS CMD set to 01, the PART ATTRIBUTES first_packet, next_packet, last_packet have to be set.

Certain situations (change of session or parallel DDL activities) may require another parsing of the statement. Upon execution, this is signalled by SQLCODE -8. As SAP DB knows the statement (see PREPARE in REQUEST SEGMENT HEADER), the parsing is done internally. In this case, SQLCODE is set to -9 and the result is the same as after PARSE. The newly given parse ID and the new PARAMETER_INFOs in the part with PART KIND shortinfo must be used for another request having the MESSAGE TYPE EXECUTE. Only then, the statement will be executed. The old parse ID and the old PARAMETER_INFOs are no longer of any use. The old parse ID will not even be known any longer by the SAP DB task.

PUTVAL

```
*IN  - PART KINDs
            longdata

*OUT - PART KINDs
            errortext
               OR
            longdata
```

In a program, INSERT and UPDATE statements can be specified which will give LONG columns new values. This functionality can only be used when statements are executed by using PARSE and then EXECUTE.

For LONG columns, the new values can only be given as 'NULL' in the statement string or via a parameter. Only for short LONG columns in an INSERT statement, its values can be specified in the statement string using string literals.

When an INSERT or UPDATE statement is parsed, PARAMETER_INFOs and a parse ID are given back as described in the chapter PARSE. For the LONG column(s) specified in this statement using parameter(s), PARAMETER_INFO(s) are given which describe one block of 40 bytes (plus UNDEFSIGNAL) per LONG column parameter.

During EXECUTE, the values for non-LONG columns must be specified as ususal. If the LONG column is set to NULL, the UNDEFSIGNAL of the block must be hexadecimal 'FF'. The block itself may have any value.

If the LONG column is to receive a real value, the UNDEFSIGNAL should be set to blank or binary zero.

If a description block is to be given, some parts must be used according to the description in PUTVAL, Communication Protocol.

Together with LONG columns, no INSERT..SELECT is allowed.

UPDATE statements are only allowed if they update only one record.

The description blocks of one INSERT or UPDATE statement together with its PUTVAL messages are of no use for any following statement and should not be saved.

PUTVAL, Communication Protocol

With one INSERT or UPDATE statement containing LONG column values there are several messages needed from application process to SAP DB task and vice versa:

1. **PARSE**
2. **EXECUTE**
3. **0-n times PUTVAL**

PARSE

The SQL statement (the INSERT for example) is parsed as usual, using parameters for all columns in this table. Let us assume a table with (very unusual) 8 LONG columns and several other non-LONG columns.

All parameter are described in the part with PART KIND shortinfo. Even for those 8 LONG column a description is given where to put the description block for each LONG column. There is no description where to put the LONG values itself.

EXECUTE (1)

For all non-LONG columns there values have to be filled in the variable part of a part with PART KIND data at the position given with BUF POS and filled to its IN/OUT-LENGTH.

For the LONG column values we think of the following example situation:

LONG1	one short LONG value of 817 bytes
LONG2	has to be set to NULL
LONG3	a LONG value of 34 bytes
LONG4	a LONG value of 372 456 bytes
LONG5	has to be set to NULL
LONG6	a LONG value of 87 bytes
LONG7	a LONG value of 52 321 bytes
LONG8	a LONG value of 120 bytes

For all LONG columns the descriptor block has to be given in the variable part of a part with PART KIND data at the position given with BUF POS.

For LONG2 and LONG5, those LONGs which will be NULL afterwards, the UNDEFSIGNAL has to be hexadecimal FF. The rest of the descriptor block is of no use, but should be filled with hexadecimal 00.

For those LONGs becoming real values it has to be decided, if those values should be (partly) given to the SAP DB task during EXECUTE or if all LONG data will be sent using PUTVAL. Usually the LONG data is (partly) sent via EXECUTE to prevent extra communication.

It can be seen that not all data of our 6 LONG columns will fit into one message. Therefore we will need PUTVALs. But in our example we will start sending data using EXECUTE.

With EXECUTE the first byte (no UNDEFSIGNAL is given for LONG values) of a LONG value may start just behind the last non-LONG column value.

For EXECUTE and PUTVAL we have to pay attention to the rules for LONG column values which is described in a special chapter, see Rule for LONG Column Handling.

Rule for LONG Column Handling

For each EXECUTE and PUTVAL/GETVAL segment there is one strict rule:

1. The first X LONG descriptor blocks found in the data part describe LONG columns given with its complete value (VALMODE=01(vm_alldata)). 0 <= X <= N, were N is the number of non-NULL LONG column values.

2. Following is 0 or exactly 1 LONG descriptor (0 <= P1 <= 1) describing a LONG column whose value was partly sent and whose last part is in this segment (VALMODE=02 (vm_lastdata)), i.e. after this segment the whole LONG value is known in the SAP DB task. With EXECUTE no such LONG descriptor can exist.

3. If P1 = 1, the following Y LONG descriptor blocks found in the data part describe LONG columns given with its complete value (VALMODE=01(vm_alldata)). 0 <= Y. With EXECUTE no such LONG descriptor can exist.

4. Following is 0 or exactly 1 LONG descriptor (0 <= P2 <= 1) describing a LONG column whose value is partly sent and will not be sent to the full with this segment (VALMODE=00 (vm_datapart)).
   If P1 = 1, with this segment only the first part of the value can be sent.
   If X = 0 and P1 = 0, some part out of the middle of the value may be sent.

5. Following are those Z LONG descriptors describing LONG columns for whom no value is sent with this segment (VALMODE=03 (vm_nodata)).

With EXECUTE all (N) LONG column descriptors have to be filled (X+P1+Y+P2+Z = N).

With PUTVAL only those LONG descriptors have to be sent to the SAP DB task for whom values are sent in the current segment.

To say it in short: after each EXECUTE and PUTVAL only 0 or 1 LONG column value may be sent partly with VALMODE=00 (vm_datapart).

The first LONG descriptor sent with VALMODE=00 (vm_datapart) or VALMODE=03 (vm_nodata) will stop the LONG writing.

If with one EXECUTE/PUTVAL one LONG value was sent with VALMODE=00 (vm_datapart) X should be 0 with the next PUTVAL. This is not mandatory but desirable.

The SAP DB task will behave accordingly in case of EXECUTE and GETVALs.

EXECUTE (2)

Imagine the non-LONG column values uses bytes 1 – 480 of the variable part of the part with PART KIND data **à** BUF_LEN = 480. Let BUF_SIZE be 30 112.

LONG1 and LONG3 will fit in this part, the next LONG (LONG4) does not, therefore X = 2, P2 = 1, Z = 3.

The LONG column values should start directly behind the last non-LONG value. The LONG values should be directly behind each other **à** VALPOS in descriptor for $LONG_{(n+1)}$ should be VALPOS + VALLEN of descriptor in $LONG_{(n)}$.

Remember the LONG values:

LONG1	one short LONG value of 817 bytes
LONG2	has to be set to NULL
LONG3	a LONG value of 34 bytes
LONG4	a LONG value of 372 456 bytes
LONG5	has to be set to NULL
LONG6	a LONG value of 87 bytes
LONG7	a LONG value of 52 321 bytes
LONG8	a LONG value of 120 bytes

For our example the 6 non-NULL LONG column descriptors will look like this:

	INTERN POS	VALMODE	VALPOS	VALLEN
LONG1	1	All data 01	481	817
LONG3	1	All data 01	1298	34
LONG4	1	Part of data 00	1332	(*) 28776
LONG6	1	No data 03	0	0
LONG7	1	No data 03	0	0
LONG8	1	No data 03	0	0

(*) There are 28 781 bytes left according to BUF_SIZE. But according to the rule that for every part BUF_LEN MOD 8 = 0 is true, only 28 776 bytes can be used for the first part of LONG4.

In the result for this EXECUTE (beside other parts) there will be a part of PART KIND longdata. It contains all LONG column descriptors (including UNDEFSIGNAL) for non-NULL LONG values, whose writing was not finished, i.e. for which VALMODE <> 01 (vm_alldata), i.e. which do not belong to count X. The ARGUMENT COUNT of this part will show the number of LONG descriptors in this part.

All descriptors were changed in the SAP DB task, therefore these returned descriptors have to be used for all subsequent PUTVALs for this statement.

In our example a descriptor for LONG4 with some changes in it and the increased INTERN POS (set to 28777) plus the changed descriptors for LONG6, LONG7, LONG8 are returned. They are returned one directly behind the other **à** (4*41 bytes) 164 bytes are sent back after the EXECUTE of our example.

PUTVAL for LONG4

Several PUTVAL messages have to be sent to sent all data for LONG4, i.e. for the LONG column in the open state.

With PUTVAL a part of PART KIND longdata has to be sent, not PART KIND data.

Only the LONG descriptor for LONG4 (together with its UNDEFSIGNAL) has to be sent. Directly following (VALPOS=42) the LONG value has to start.

INTERN_POS will be changed by the SAP DB task. The application process is not allowed to change it after the first initialization with 1.

VALMODE has to stay with 00 (vm_datapart) for the next 11 PUTVALS (372456-28776) DIV 30600, if we assume 30600 bytes available for the LONG column value with each PUTVAL message. Let us assume BUF_SIZE = 30648. 41 Bytes are used for the descriptor. 30607 are not used. The maximum value x for which (x MOD 8 = 0) is true, is 30600.

VALMODE = 00 (vm_datapart), VALPOS = 42, VALLEN = 30600 in our example.

For each PUTVAL the changed and returned LONG descriptor of the PUTVAL/EXECUTE before has to be used.

The ARGUMENT COUNT in the longdata part has to be set to the number of LONG descriptors written in this part.

If no LONG value is in an open state, no longdata part will be returned with the PUTVAL.

In our example the descriptor for LONG4 will be returned.

Last PUTVAL for LONG4

There are some (in our example: 7080) bytes left out of the value of LONG4. For them VALMODE in the corresponding LONG column descriptor has to be changed to 02 (vm_lastdata). Every LONG column sent with several PUTVALs has to use VALMODE = 02 (vm_lastdata) with the PUTVAL the last data-portion of this LONG column is sent to the SAP DB task.

VALMODE = 02 (vm_lastdata), VALPOS = 42, VALLEN = 7080.

It can be seen (assuming BUF_SIZE = 30648 as before), that the whole LONG6 (87 bytes) and the first part of LONG7 (> 53KB) will fit in the longdata part (BUF_LEN = 7121).

In every PUTVAL in which more than one LONG column is handled, the variable part of the longdata part should look like this:

1. First LONG descriptor with VALPOS=42
2. starting at position 42: data belonging to first descriptor with the length given in VALLEN
3. directly behind the first value the second descriptor with VALPOS = (2*41)+VALLEN of first descriptor+1

4. directly behind the second descriptor the corresponding value
5. third descriptor….

To say it shortly:
Always write a LONG descriptor and the corresponding value directly behind each other and directly behind the combination descriptor/value written before.

In our example the descriptor for LONG6 (that one returned with EXECUTE) has to be written to position 7122 and has to look like this:

VALMODE = 01(vm_alldata), VALPOS = 7163 (7122+41 bytes of descriptor),
VALLEN = 87.

The value for LONG6 has to start at position VALPOS.

The descriptor for LONG7 will start at (7163+87=7250) and look like this:

VALMODE = 00 (vm_datapart), VALPOS = 7291 (7250+41 bytes of descriptor),
VALLEN = 23304
(Assuming BUF_SIZE 30648: VALLEN (30648-7290=23358 **à** 23352 MOD 8 = 0))

No LONG descriptor for LONG8 needed.

Last PUTVAL for the Statement

The rest of the value of LONG7 has to be handled like the rest of LONG4, i.e. some PUTVALs with VALMODE = 00 (vm_datapart), the last part of the value with VALMODE = 02 (vm_lastdata). In our example the whole rest of LONG7 will fit into one message, therefore only one PUTVAL for LONG7 is necessary, which will use VALMODE = 02 (vm_lastdata).

As with LONG6 behind the rest of LONG4, LONG8 will fit behind the rest of LONG7. Therefore the longdata part looks like this:
Descriptor for LONG7, rest of LONG7, descriptor for LONG8
(VALMODE = 01,vm_alldata), LONG8.

But it is the last PUTVAL for the whole statement (in our example, the INSERT). That makes things different.

The SAP DB task does not remember how many LONG descriptors belong to this statement. Therefore the SAP DB task is not able to decide if the last PUTVAL belonging to this statement was done. On the other hand, for the SAP DB task this statement is in an open state. If an error occurred during any PUTVAL, the whole statement with INSERT of non-LONG values and all PUTVALs has to be rollbacked. For this reason, an implicit subtrans is opened for the statement. Therefore the SAP DB task has to know, when the statement is finished, i.e. when the last PUTVAL is done. Then the task can ‘close’ this statement, i.e. ends this subtrans.

For this reason, the last PUTVAL for a statement has to send an extra and special LONG descriptor. In this LONG descriptor only the fields VALMODE and INFO SET are of any use. Therefore any of the old LONG descriptors can be copied. VALMODE has to be set to

05 (vm_last_putval). This LONG descriptor has to be the last in the longdata part. No data belongs to this descriptor.

Even if there is not enough space in the last PUTVAL message containing real values, this special LONG descriptor has to be sent. Then another PUTVAL message has to be sent with exactly this special LONG descriptor in it.

If all LONG values fit in the EXECUTE, no extra LONG descriptor with VALMODE = 05 (vm_last_putval) is allowed. Only if at least one PUTVAL was necessary, this extra LONG descriptor is needed.

Error

In case the application / the application process found an error, for example a file reading error during LONG PUTVALs, i.e. an error not found by the SAP DB task, then it has to work as follows:

For the SAP DB task everything seems to be fine. There is possibly one LONG column in an open state which belongs to an open statement. The LONG column which is the open state is not known by the SAP DB task.

Therefore the application process has to send a PUTVAL with the descriptor of the LONG column in open state with
VALMODE = 02 (vm_lastdata), VALPOS = 42, VALLEN = 0.
Even VALPOS and VALLEN are important here.

The extra LONG descriptor with VALMODE = 05 (vm_last_putval) has to be appended at position 42.

If no LONG value is in an open state, only the extra and special LONG descriptor with VALMODE = 05 (vm_last_putval) has to be sent.

MASS CMD = 01

If more than one row will be handled at once (MASS CMD set to 01), then for those rows using LONG columns the following communication protocol exists:

All data of one row has to be sent together, i.e. the values for non-LONG columns inclusive the LONG descriptors and directly following the LONG values for this row.

If for one row all values inclusive those for LONG column fit in one message, the next row can be sent in the same message with its LONG values and another one and so on. All will be sent with EXECUTE.

At the end there will be one of three situations:

1. all row values (incl. LONG column values) fit in one EXECUTE, that is the only message
2. the last row (non-LONG column values + LONG column values) will not fit as the whole in this message, the non-LONG columns fit, but one or more LONG values do not fit
3. the last row will not fit, because not even all of the non-LONG values fit

With situation 1, there is no problem, the message can be sent.

In situation 3 the last row whose non-LONG values do not fit in the current message will not be written in this message. Non-LONG columns have to stay together. Therefore the current message will not be filled up to the last byte, but sent now. And the row which did not fit will be the first one in the next message send with EXECUTE.

Situation 2 is a little bit more complicate. In case only few LONG value bytes can be sent together with (short) non-LONG values, it may be of use to behave as in situation 3 if this row together with its LONG values will fit in the next EXECUTE message.

Otherwise PUTVALs have to be used. That means, x full rows + the non-LONG column values of the current row and some of its LONG values (or part of it) are sent with the current EXECUTE message. Then some PUTVAL messages have to be sent with the rest of the LONG column values of this current row. The last PUTVAL for this row has to sent an extra and special LONG descriptor with VALMODE = 05 (vm_last_putval), last data for this row.

With this PUTVAL, no matter how much space is left, no non-LONG values may be sent.

Therefore the next row to be sent will start with a new EXECUTE message. For this one the rules given above are true: if one row fits as the whole, one or more others may be in the same data part, use PUTVAL, if necessary.

GETVAL

```
*IN  - PART KINDs
            longdata

*OUT - PART KINDs
            errortext
               OR
            longdata
```

In a program, SELECT statements can be used which specify LONG column(s) in the select list. During SELECT..INTO or FETCH, the values of these LONG columns will be given to the application process.

When a SELECT or FETCH statement is parsed, PARAMETER_INFOs and a parse ID are given back as described in the chapter PARSE. For the LONG column(s) specified in the select list, PARAMETER_INFO(s) are given describing one block of 40 bytes (plus UNDEFSIGNAL) per LONG column. When a SELECT or FETCH statement is used together with the MESSAGE TYPE DBS and the field WITH INFO has the value 01, then the part of PART KIND shortinfo contains a description of one block of 40 bytes (plus UNDEFSIGNAL) per LONG column.

During EXECUTE, the values for non-LONG columns are written as usual. For a LONG column, the description block is given at the BUFPOS position specified in the corresponding PARAMETER_INFO.

If the LONG column has the NULL value, the UNDEFSIGNAL of the block is hexadecimal FF. The block itself is then of no interest. No real LONG column value exists.

If the LONG column has a real value, the UNDEFSIGNAL is set to blank or binary zero.

SAP DB will fill the first 27 bytes of the block and the application process is not allowed to change these bytes except in a special case described below.

All LONG column values are given behind all non-LONG columns and description blocks. If a description block is given, some parts will be used according to the following description.

VALPOS specifies the first byte of the value of the LONG column in the variable part of the message. The value has no UNDEFSIGNAL in front.

VALLEN specifies the length of this portion of the LONG column value; i.e., the length of the LONG column value as it is in the variable part of this segment.

VALMODE specifies how this data portion is to be interpreted:

03 (vm_nodata) In this message, there is no portion of this LONG column because there is not enough space.
01 (vm_alldata) This data portion is the whole value of the LONG column.
00 (vm_datapart) This data portion is one of several portions. The LONG column value must be cut in several portions because the space in one message is too small for the whole

LONG column value.
The other data portions will be given during following messages using GETVAL.

06 (vm_data_trunc) This data portion is not the whole (rest of the) LONG value. According to VALLEN given with this LONG descriptor, the application process told the SAP DB task how much space it has available for this LONG value. The LONG value is longer. Therefore it is truncated to fit into this available memory space.

The values of the LONG columns are given in the order in which the description blocks are written.

In Rule for LONG Column Handling it is written, which VALMODEs will be found in the result of EXECUTE and GETVAL messages.

If with EXECUTE at least one LONG descriptor with VALMODE = 00 (vm_datapart) or = 03 (vm_nodata) is returned to the application process, a loop of messages must start using GETVAL to read the missing LONG column values or portions from the SAP DB task.

Those LONG descriptors returned with EXECUTE or GETVAL have to be used for the next GETVAL. During this loop, the UNDEFSIGNAL of a block should not change and the first 28 bytes of a block are not allowed to be changed by the application process (except in a special case described below).

With GETVAL, the part of PART KIND longdata must be filled with all blocks given that have no UNDEFSIGNAL = 'FF' and no VALMODE = 01(vm_alldata), 02 (vm_lastdata) or 06 (see below for a special case); i.e., all blocks describing LONG column values that were not completely given by the SAP DB task yet. The first LONG descriptor should be that one which was returned with VALMODE = 00 (vm_datapart) during the GETVAL/EXECUTE before. LONG descriptors returned with VALMODE = 06 are the only ones which are allowed to be sent before that one with VALMODE = 00 (vm_datapart).

VALPOS is of no interest in the request of GETVAL

VALLEN has to be set to the number of bytes left in the host variable to store the following data portions of the corresponding LONG column value. SAP DB will use this value and will not send back more bytes than the number of bytes that can be stored.

If there are more bytes in the column value but these are not given back because of the smaller VALLEN, VALMODE will be set to 06 for the output. In this case, INTERNPOS will be set to (length-of-the-whole-LONG-column-value + 1); i.e., to one byte more than the length the host variable should have had.

In case of VALMODE = 06, the LONG value is in no open state, that means: further GETVALs for this LONG value are not allowed. If for some reason, it is necessary to read smaller pieces out of a LONG value one after each other, it is necessary to prevent the SAP DB task from closing the LONG value. This has to be done with no_close set in INFO SET for each GETVAL for this LONG value. The VALMODE for this LONG descriptor has to be changed to 00 (vm_datapart) and no_close set in INFO SET for each subsequent GETVAL for this LONG value. Even in case of the last data portion returned (VALMODE = 02 (vm_lastdata)), the LONG value is not closed. It has to be closed explicitly by sending a GETVAL message with VALMODE = 07 for this LONG descriptor. This

explicit closing has to be used for such a LONG descriptor used with no_close set in INFO SET no matter if it is read to its end or not.

As a result of GETVAL, the part of PART KIND longdata consists of a LONG descriptor (inclusive UNDEFSIGNAL) and the corresponding value directly behind its descriptor and this combination directly behind the combination descriptor/value written before.

At the end those LONG descriptors can be found of which no value is in this part.

ARGUMENT COUNT will be set to the number of LONG descriptors in this part.

The description blocks of a statement together with its GETVAL messages are of no use for any following statement and should not be saved.

In case of an error not occuring in the SAP DB task, but in the application (file error in the file the LONG column is put to, for example), those LONG descriptors which are in open state (VALMODE = 00 (vm_datapart) or VALMODE = 06 and no_close was set for it), have to be closed. This can be done with VALMODE = 00 (vm_datapart), VALLEN = 0 and , of course, no_close not set or with VALMODE = 07 (close).

In case of a statement with MASS CMD set to 01, the layout of the result message of a reading statement (SELECT or FETCH) is different to the layout of a request message of a writing statement as INSERT or UPDATE. With the EXECUTE of an INSERT or UPDATE, the whole row inclusive all LONG columns has to be sent to the SAP DB task before the first byte of the second row has to be sent.

With EXECUTE of a SELECT or FETCH, the non-LONG column values of as many result rows as exist or as wanted or as will fit in the message are filled into the message. If there is some space left, LONG values are filled behind the last non-LONG-column-result row-parts. This is done in the sequence the LONG descriptors are found in the data part. To get the rest of the LONG values, several GETVALs have to be performed.

That means: to output one full result row inclusive all of its LONG values, sometimes, the result of the EXECUTE has to be stored somewhere in the application process and the GETVALs needed to get the wanted LONG values have to be performed. Therefore the number of wanted result rows for one EXECUTE of FETCH should be small enough to be able to store its non-LONG column values temporarly.

HELLO

```
*IN - no part

*OUT - no part
```

This MESSAGE TYPE allows a user to verify whether the SAP DB task assigned to this user is still able to process statements--without the user having to issue a SAP DB statement. At the same time, this prevents the session from being ended implicitly as a result of the user's inactivity. This is the only MESSAGE TYPE without any parts.

SENDING AND RECEIVING MESSAGE BLOCKS

If an application process has allocated a message block that is not longer than the maximum length returned by SQLACONNECT, this message block is used for both directions of communication between application process and SAP DB task.

For the direction to the SAP DB task, the first segment, i.e. the first command, must be directly behind the MESSAGE HEADER.

When the results are returned, during SQLARECEIVE the application process obtains a pointer to the message block being returned. The pointer of the return-message-block differs from the pointer of the request-message-block. The pointers to the return-message-blocks of different results may be different from each other. Therefore, the request pointer must be saved and the result pointer be forgotten after interpreting the return-message-block to which it belongs.

In some cases, e.g., the parsid in the MESSAGE TYPEs PARSE and EXECUTE, parts returned must be sent to the SAP DB task again. It must be ensured that PART ATTRIBUTES, SEGMENT OFFSET and BUF SIZE are changed between the part returned and the part belonging to the request.

SAP DB FUNCTIONS FOR SYSTEM PROGRAMMERS

Only some of the SAP DB statements described in this section (like DECLARE CURSOR, OPEN, CONNECT, CREATE USER, and ALTER PASSWORD) are available to users of the interactive SAP DB interface or to programmers working with the SAP DB system via SAP DB statements embedded in programming languages.

DECLARE CURSOR, OPEN

In a program, DECLARE CURSOR is specified in a declare section. The corresponding OPEN is specified in that part of the program that is dynamically executed. If one checks the trace information written by SAP DB, no OPEN will be found. The application process (i.e., the precompiler) must store the DECLARE CURSOR and send it to SAP DB when the corresponding OPEN is found in the program. The DECLARE CURSOR will then search for the result rows and will not wait for an OPEN statement to be started.

FETCH with MASS CMD set to 01

This statement basically functions in the same way as the FETCH statement described in the SAP DB Reference Manual. Whereas each normal FETCH only provides a single result row plus a description, if requested, of the output columns (WITH INFO = 01), FETCH with MASS CMD set to 01 passes several result rows to the user--or, i.e., as many rows as are wanted respectively as many rows as will fit in the message block together with the description of the output columns.

Lock handling will be done as with FETCH.

FETCH FIRST provides the first result rows, FETCH LAST the last, FETCH PREV the previous ones and FETCH POS the result rows following the specified position. Even FETCH LAST and FETCH PREV provide the result rows in ascending FETCH order (i.e., just like the normal FETCH FIRST, FETCH NEXT and FETCH POS).

If FETCH SAME is called following FETCH with MASS CMD set to 01, then it provides a result row with the largest key contained in the last FETCH set (regardless of FETCH NEXT or PREV).

In a part of PART KIND resultcount, a number must be sent to the SAP DB task specifying the number of hits to be provided. The returned segment then contains a part of PART KIND shortinfo and one of PART KIND columnnames (if requested by the field WITH INFO = 01 and the MESSAGE TYPE DBS), a part of PART KIND resultcount and a part of PART KIND data. The only difference to a FETCH statement is that ARGUMENT COUNT in the data part may be larger than 1.

FETCH with MASS CMD set to 01 is not allowed for SELECTs in which FOR UPDATE was specified.

SELECT, DECLARE with MASS CMD set to 01

These statements basically function in the same way as SELECT and DECLARE CURSOR. With SELECT or DECLARE, one value can be specified for each parameter given in the statement, in the following called 'data-packet'.

With SELECT and DECLARE CURSOR with MASS CMD set to 01, several data-packets can be specified, each containing one value for each of the parameters.

The result is one (!) result table in which all the result rows specified with the first data-packet plus the result rows specified with the second data-packet plus ... plus the result rows specified with the last data-packet are stored.

Several message blocks can be used to build one result table with several data-packets.

SELECT and DECLARE CURSOR with MASS CMD set to 01 can only be used together with the MESSAGE TYPEs PARSE and EXECUTE. While parsing, everything looks as usual, except that MASS CMD is set to 01 instead to its usual value 00. If there is no set function (AVG, MIN and so on), no GROUP BY, no HAVING, no UNION, no EXCEPT and no INTERSECT specified in the statement, this statement can be handled as a mass command.

This will be shown in the parse ID with application info1 set to 114 (select_with MASS CMD found), 115 (for_upd_select_with MASS CMD found), 116 (reuse_select with MASS CMD found) or 117 (reuse_select_for_update with MASS CMD found).

If this statement can only be handled like a normal SELECT or DECLARE CURSOR, application info1 will have the value 44, 45, 46, 47 respectively, i.e. the above value minus 70. Then no more than one data-packet is allowed and the statement must be executed as described in the chapter EXECUTE.

The application process is not allowed to change 44/45/46/47 to 114/115/116/117. SAP DB checks for such a misuse and returns an error.

If application info1 has the value 114, 115, 116 or 117, one or more data-packets can be given during execution. While executing, the part of PART KIND parsid looks as usual. If the statement was parsed with MASS CMD set to 01, but only one data-packet belongs to this statement, the statement should be executed like a normal SELECT (see EXECUTE) and with the application info1 set to 44/45/46/47 (old application info1 minus 70 (mass_statement)). If application info1 has the value 114/115/116/117 and more than one data-packet is to be given, the parsid part consists of the unchanged parse ID.

Another part (PART KIND resultcount) consists of a number in the format Fixed(10) specifying the number of results found during the last messages belonging to this statement. For the first message in one statement, the UNDEFSIGNAL of this value must be 'FF'. For the following messages, the resultcount given with the previous message must be given to SAP DB.

The ARGUMENT COUNT of the data part states the number of data-packets actually given to SAP DB. If this is the last message for this statement, last_packet must be set in PART ATTRIBUTES. If this is the first one first_packet has to be set, for those in the middle next_packet has to be set.

Every data-packet looks like a data-packet for the corresponding SELECT or DECLARE CURSOR statement, this means, as described in the PARAMETER_INFOs. All data-packet have to be sent directly one behind the other.

The result looks like a result of a normal SELECT or DECLARE CURSOR.

SELECT ... INTO with MASS CMD set to 01

This statement basically function in the same way as SELECT ... INTO usually does. With SELECT ... INTO, one value can be specified for each parameter given in the statement, in the following called 'data-packet'.

With SELECT ... INTO with MASS CMD set to 01, several data-packets can be specified, each containing one value for each of the parameters. The result is one result row for each of the specified data-packets. In case no result exists for one data-packet a result row filled with x'FF' is returned.

SELECT ... INTO with MASS CMD set to 01 can only be used together with the MESSAGE TYPEs PARSE and EXECUTE. While parsing, everything looks as usual, except that MASS CMD is set to 01 instead to its usual value 00.

INSERT, UPDATE, DELETE with MASS CMD set to 01

These statements basically function in the same way as usual, except for INSERT-SELECT, which is not allowed together with MASS CMD set to 01.

With these statements, it is possible to INSERT or UPDATE or DELETE several rows during one call with EXECUTE. These statements are only allowed together with PARSE and EXECUTE, not with DBS.

They are parsed as usual, MASS CMD is set to 01 instead of the usual value 00. If there is no INSERT-SELECT specified in the statement, this statement can be handled as a mass command. This will be shown in the parse ID with the application info1 set to 70 (mass_statement).

If this statement can only be handled like a normal INSERT, UPDATE or DELETE, application info1 will have the value 0 (none). Then no more than one data-packet is allowed and the statement must be executed as described in the chapter EXECUTE.

The application process is not allowed to change 0 to 70. SAP DB checks for such a misuse and returns an error.

If application info1 has the value 70, one or more data-packets can be given during execution. If the statement was parsed with MASS CMD set to 01 with only one data-packet belonging to this statement, the statement should be executed like a normal INSERT/UPDATE/DELETE (see EXECUTE) and with application info1 set to 0 (old application info1 minus 70 (mass_statement)).

If application info1 has the value 70 and more than one data-packet is to be given, the parsid part consists of the unchanged parse ID.

Another part (PART KIND resultcount) consists of a number in the format Fixed(10) specifying the number of rows handled during the last messages belonging to this statement. For the first message in a statement, the UNDEFSIGNAL of this value must be 'FF'. For the following messages, the resultcount given with the previous message must be given to SAP DB.

ARGUMENT COUNT of the data part states the number of data-packets actually given to SAP DB.

If this is the last message for this statement, bit 1 must be set in PART ATTRIBUTES of the data part. If this is the first one first_packet has to be set, for those in the middle next_packet has to be set.

Every data-packet looks like a data-packet for the corresponding INSERT, UPDATE or DELETE statement, this means, as described in the PARAMETER_INFOs. All data-packet have to be sent directly one behind the other.

The result looks like a result of a normal INSERT, UPDATE or DELETE.

If SQLCODE <> 0, then ERRORPOS specifies the number of the data-packet producing this error and a part of PART KIND errortext is returned. Another part with PART KIND resultcount is returned to show the number of changed rows up to the wrong data-packet.

In case of an error, all changes done before this error occurred remain done. No changes are done for the wrong data-packet and the data-packets following the wrong one.

CONNECT

In case of a CONNECT, a part of PART KIND data must be filled with the name of the user (UNDEFSIGNAL plus name with 64 bytes) if the name of the user was not given in the SQL statement. Then the password in a crypted format (UNDEFSIGNAL plus 24 bytes) and the result of SQLTERMID (UNDEFSIGNAL plus the 18 bytes of the result) must be specified.

CREATE USER with crypted password

Usually, the <create user statement> is given like this: CREATE USER <user name> PASSWORD <password> ...

In this case, the password is sent to SAP DB in a readable format. As someone would like to hide the <password> even in the message, the crypted format of the password can be sent.

The statement then looks like this: CREATE USER <user name> ... without 'PASSWORD <password>'.

The <password> in its crypted format (24 bytes) must be given in a part of PART KIND data together with an UNDEFSIGNAL.

This format can be used together with the MESSAGE TYPEs DBS, SYNTAX and PARSE.

ALTER PASSWORD with crypted password

Usually, the <alter password statement> is given like this:
ALTER PASSWORD <old password> TO <new password>
  or
ALTER PASSWORD <user name> <new password>

In these cases, the new password is sent to SAP DB in a readable format. As someone would like to hide the <new password> even in the message, the crypted format of the new password can be sent.

The statement then looks like this:
ALTER PASSWORD <old password> TO
  or
ALTER PASSWORD <user name> without '<new password>'.

The <new password> in its crypted format (24 bytes) must be given in a part of PART KIND data together with an UNDEFSIGNAL.

This format can be used together with the MESSAGE TYPEs DBS, SYNTAX and PARSE.

SET LANGUAGE

Upon CONNECT, the SAP DB task does the language setting valid for the current user. This language setting is used for error messages and for the names of months in SQLMODE ORACLE and its data type date. The language used is english.

During the SAP DB session, SET LANGUAGE allows the user to switch the language used. The desired language is entered as CHAR(3) value plus UNDEFSIGNAL in a part of PART KIND data. SET LANGUAGE is neither parsable nor executable, and can only be issued with the MESSAGE TYPE DBS.

At present only english messages and names of months are known in the SAP DB task. Therefore it is not possible to set the language to another value than english.

SET FORMAT

As default, every user uses the date-and-time format specified during the configuration.

If a user uses an application which uses ODBC or JDBC for communication with the SAP DB task, then the date-and-time format called ANSI is used, no matter of the format specified during configuration.

This format is standardized by ODBC-commitees and looks like this for a TIMESTAMP value:

```
'YYYY-MM-DD HH:MM:SS.MMMMMM' '1999-01-23 14:30:08.456234'
```

For DATE and TIME values it looks like the format JIS, described in the reference manual.

When an application is used, which uses ODBC or JDBC for communication with the SAP DB task, then no SET FORMAT statement may be executed.

With other applications, the user can specifiy SET FORMAT to change the format used for this session.

SET FORMAT ISO
SET FORMAT EUR
SET FORMAT USA
SET FORMAT JIS
SET FORMAT INTERNAL
SET FORMAT ANSI
SET FORMAT ORACLE

are available. For a description of the different formats, see the reference manual. The format ORACLE results in ‘DD-MON-YY’ for values of datatape TIMESTAMP respectively those columns being defined with DATE using SQLMODE ORACLE. For values of datatypes DATE and TIME the format ORACLE looks like the INTERNAL format.

The date-and-time format is changed user- and session-specifically. This means, the default date-and-time format will be used again after another CONNECT.

SET FORMAT is neither parsable nor executable and can only be issued with the MESSAGE TYPE DBS.

SET ISOLATION LEVEL <unsigned integer>

With this command, the isolation level of the session can be changed for all subsequent commands in this session. All known isolation level can be specified.

SET NLS_SORT

SET NLS_SORT BINARY/<identifier> This statement tells the SAP DB task how to sort resultsets in the current session, if the order by clause is applied to alphanumeric columns. If BINARY is specified, the normal binary sorting is used. Otherwise <identifier> identifies a MAPCHAR SET which defines the sorting rules.

SET TIMEOUT ON/OFF

This command is no user-command, that means: the command can only be used internally with PRODUCER <> user.

SET TIMEOUT … is neither parsable nor executable and can only be issued with the MESSAGE TYPE DBS.

With this command the command timeout of this session can be switched on (as it is as default) or off.

SET SESSION <sessionno> TRACE LEVEL <no>

Sometimes one would like to know what a running application is just doing. This is often necessary in case this application seems to run endless or much to slow. In this case it is not possible to stop the application, switch on some trace for it and rerun the whole application.

It is necessary to start some trace for the running application. This can only be done by another session.

With this command, session x can specify, that another session, i.e. the session with the number sessionno should return the specified trace level no to its application program. Each application program has its own defintion, what kind of trace will be written together with the different trace levels.

For a description of the different trace levels of different applications see their manuals.

EXPLAIN

With EXPLAIN, there are some options to get more information as with the usual EXPLAIN.

EXPLAIN VIEW SELECT ...

In case there is a joinview in the from-part of this SELECT, usually only the name of this joinview is given for all 2 to 16 steps of performing this join. This is done because of privilege handling. Not all users allowed to see this joinview are allowed to see the underlying tables or views. Therefore they are not told how this joiniew was specified and which tables or views are needed for it.

If EXPLAIN VIEW is given instead of EXPLAIN, the names of the underlying tables or views are given as result. This allows the user to see in which sequence the underlying tables or views are used.

EXPLAIN JOIN SELECT ...

Sometimes it is useful to see the values which were used and calculated for each join-performing step. Therefore for each step values like STRATEGY, OWNER, TABLENAME (as with the normal EXPLAIN) are given together with EXPECTED_ROWS, MULTIPLIER, REVERSE_MULTIPLIER, NEW_LEFT_SIZE, NEW_LEFT_ROWS and ACCUMULATED_COSTS.

EXPLAIN SEQUENCE SELECT ...

Sometimes it is useful to see the intermediate information prepared to find the best join sequence for a SELECT statement. This info can be seen with EXPLAIN SEQUENCE.

The result is divided into several parts:

- one table access
- possible join strategies from one table to another
- possible join strategies from x tables to another one
- info about the strategy used for sequence search
- calculated costs for different sequences of all tables

one table access

In this part for each table implicitly or explicitly specified in the FROM part of the statement it is shown, which access strategy would be used if this table together with the corresponding predicates was given in one SELECT statement.

For each table the following info is shown:

- TABLE No.
- Tablename
- Search strategy (see in module VGG17 g17stratenum_to_line for the possible names and meanings)
- Number of pages/part of one page in which records can be found which meet the conditions given in the predicates

- Number of pages which have to be checked for finding all pages in which records could be found which meet the conditions given in the predicates
- Pages in this table
- Access cost for access all pages needed
- Estimated number of records per page

In an example it looks like this (example taken from EXPL.vdnts):

```
TABLE No.  1
T
  SINV IN
 pages_found 0.100E-06
 strat       0.100E-06
 all_pages         1
 cost              1
 recs_per_p        1
-----------------------------------------------------------------------
TABLE No.  2
T1
  KEY RANGE
 pages_found 0.400E+00
 strat       0.100E+01
 all_pages        40
 cost             41
 recs_per_p      250
-----------------------------------------------------------------------
TABLE No.  3
T
  KEY EQ
 pages_found 0.100E+01
 strat       0.100E+01
 all_pages         1
 cost              1
 recs_per_p        1
```

possible join strategies from one table to another

In a table it is specified which join strategy could be used if the table with number TO was joined to the table with number FROM.

See in module vak681 a681tr_joinvals for the possible join strategies:

```
            to_single_keyfield := 'SK'
            to_unique_field    := 'UI'
            to_keypart         := 'KP'
            to_key             := 'MK'
            to_first_keyfield  := 'FK'
            to_invfield        := 'I'
            to_all_invfields   := 'MI'
            to_invpart         := 'IP'
            to_mt_join         := 'MJ'
            to_eq_field        := '='
            to_lt_gt_field     := '><'
            to_ne_field,
            to_cartesian_prod  := '??'
            to_illegal         := '--'
```

```
-----------------------------------------------------------------
| FROM  1   2   3
TO
1       --  MJ  MJ
2       =   --  =
3       SK  SK  --
```

possible join strategies from x tables to another one

With this it is shown which join strategy couls be used if all tables given in Used tables were joined before and table TO had to be joined as the next one.

The following results will be given:

TO join to the x-th table in the from-part (or the x-th base-table, if there are joinviews in the from-part)
Via information about the possible access strategy to that table (via INDEX or KEY or so)
Multiplier the expected number of rows which will be found in table 'TO' for each row in the up-to-then intermediate result table
T how many tables have to be joined together to form the information needed for the possible access strategy given in 'via'
F how many columns are needed to form the information needed for the possible access strategy given in 'via'
J the index in an array (0-x) in which all join-conditions are stored. Be aware that because of transitive join-conditions some new ones are build internally and filled into this array
I the number of the index (multiple index) or the columnnumber (single index) which is accessed using the access stratgy given in 'via'
Used tables as many tablenumbers as 'T' specifies are given to show the tables which have to be joined together before table 'TO' could be joined if access strategy 'via' shall be used

```
-----------------------------------------------------------------
TO  via          multiplier   T  F  J  I  Used tables
1  INDEX PART     0.100E+01   1  1  0  1  [  2 ]
1  INDEX         0            1  1  0  2  [  2 ]
1  INDEX PART     0.100E+01   1  1  2  1  [  3 ]
1  INDEX         0            1  1  2  2  [  3 ]
```

Here it is shown the combination of best join strategies for all tables. Sometimes this sequence of tables is not a possible one with these strategies. Imagine a join strategy which joins to table x and needs info from table y and z. This may be the best join strategy for x. But it is not a possible one if y and z are not joined before.

This can be seen in this example where ji gives the number of the table the given join strategy for table 1 as a joined table before. Table 2 was not joined before, therefore this strategy/sequence is not a possible one. 1 is the default value for ji.
Mi gives the index in the array of tables.

```
-----------------------------------------------------------------
TO  via           multiplier    costs         ji  mi
1  INDEX         0             0.100000E+01  2   2  multiplejoin
```

```
2  EQUAL FIELD    0.200E+02     0.200391E+07  1   0
3  SINGLE KEY     0.100E+01     0.112777E+03  1   0
```

info about the strategy used for sequence search

It is the goal of a SAP DB task to find the best join strategy for all tables which are named in the from clause. On the other hand, this search is not allowed to last too long. Therefore one has to pay attention to the search cost itself. And therefore a full permutation of all table sequences is possible for a small number of tables, but not for a large number.

With SA DB there are some JOIN_SEARCH_LEVEL (see installation parameters)

9	full permutation
4	heuristic
1	greedy algorithm
0	let SAP DB task decide the level according to the number of tables

For JOIN_SEARCH_LEVEL = 0, the number of tables (N) can be specified which will use level 9, 4 or 1.

0 < N <= JOIN_MAXTAB_LEVEL9 (default 5) : use level 9
JOIN_MAXTAB_LEVEL9 < N <= JOIN_MAXTAB_LEVEL4 (default 16) : use level 4
N > JOIN_MAXTAB_LEVEL4 : use level 1

The next info says which sequence search strategy was used

JOIN SEQUENCE SEARCH = AUTOMATIC - LEVEL 9

calculated costs for different sequences of all tables

The info is given which sequence of all tables (seen in brackets) results in which costvalues. The calculation is done in the sequence shown. With < the costs are marked which are the smallest one up to that time. > shows costs which are higher than the lowest found by then.

The table sequence which is the last one marked with < is the best sequence and will be used.

Even in case of a full permutation not all combinations will be shown. In this example, the join from table 1 to table 3 is more expensive than the full sequence 1,2,3. Therefore no sequence of tables starting with 1,3 has any chance to result in smaller costs. Therefore no full sequence is calculated and written in this list.

< COSTVALUE : 0.410292E+02 [1 2 3]
> COSTVALUE : 0.194875E+03 [1 3 2]
< COSTVALUE : 0.410292E+02 [2 1 3]
> COSTVALUE : 0.194875E+03 [3 1 2]

CCCRASHHH

Testing the termination and error analysis mechanisms of the various runtime environments requires a statement which will cause SAP DB tasks to ABEND. This is realized using the statement CCCRASHHH which is only available to the SYSDBA and must be used with the MESSAGE TYPE DBS.

Tracing

Tracing in slowknl

For error and performance analysis sometimes it is necessary to output some information produced during statement handling in the SAP DB task. This can be done using the switch statement, which is written here in a reference manual-like manner:

```
<switch command> :=
  DIAGNOSE SWITCH OFF
| DIAGNOSE SWITCH MINBUF
| DIAGNOSE SWITCH MAXBUF
| DIAGNOSE SWITCH BUFLIMIT <unsigned integer>
| in version 7.3:
      DIAGNOSE SWITCH LAYER <string literal>
            [DEBUG <string literal>]
            [<switch limit>]
  in version >= 7.4
            DIAGNOSE SWITCH
       [LAYER <string literal]
       [DEBUG <string literal>]
       [TOPIC <string literal>]

<switch limit> ::= LIMIT
  START <string literal>
  [COUNT <unsigned integer>]
  [STOP <string literal>]
```

With OFF tracing is switched of.

With MINBUF/MAXBUF it can be specified if buffer output will be in hexadecimal + decimal + character form plus the position or if it will be only with hexadecimal and character form.

With BUFLIMIT it can be specified which is the maximum buffer position to be written no matter if the buffer is a longer one.

With LAYER it is specified for which layer in the SAP DB task (AK, GG, BD,...) the Pascal routines write their names in the trace when they are called.

With DEBUG it is specified for which layer or part of a layer in the SAP DB task (AC, BI, KF, GG, ...) the Pascal routines write some info about the current state of buffers, variables to the trace.

With TOPIC it is specified for all SAP DB versions >= 7.4 which C++ components have to write trace info.

With START it can be specified at which Pascal routine the tracing should start.

With COUNT it can be specified which call to this routine should start the tracing.

With STOP it can be specified at which Pascal routine tracing should stop.

This statement can only be used with the MESSAGE TYPE DBS.

Flush the trace in any kernel

The statement VTRACE forces the writing of the TRACE information of all SAP DB tasks onto the disk. This can be used when searching for a faulty system behaviour to receive the TRACE information.

This statement can only be used with the MESSAGE TYPE DBS.

DROP PARSEID

This statement deletes the parse ID--specified in the part of PART KIND data without UNDEFSIGNAL--of a statement previously issued using PARSE, as well as the pertinent information.

This statement can only be used with the MESSAGE TYPE DBS.

DESCRIBE [<result table name>]

This statement provides descriptions of all result table columns. These descriptions consist of a part of PART KIND shortinfo and a part of PART KIND columnnames (see there)

The SELECT statement generating the named or unnamed result table must have been parsed and/or executed.

DESCRIBE

All the following DESCRIBE statements are exclusively used by SAP DB LOAD respectively the Replication Manager.

These statements can only be used with PRODUCER <> user_cmd.

DESCRIBE USER

This statement is used by SAP DB LOAD if auditing is required.

A description of all users and the corresponding userids is returned.

This statement will not be available with SAP DB version >= 7.4.

DESCRIBE CONSTRAINT/DEFAULT/INDEX <table name>

These statements are used by SAP DB LOAD respectively the Replication Manager, whenever a table is extracted from or imported into the database.

If DESCRIBE CONSTRAINT is send, the datapart contains the number of the constraint to be described.
The SAP DB task returns a description of the contraint.

If DESCRIBE DEFAULT is send, the datapart contains the number of the column of the DEFAULT to be described.

If DESCRIBE INDEX is send, the datapart contains the number of the index to be described.

DESCRIBE TABLE <table name> [EXCEPT [CONSTRAINT] [RESTORE]]

This statement is used by SAP DB LOAD respectively the Replication Manager whenever a table is extracted from or imported into the database.
The SAP DB task returns a description of structure of the base table identified by <table name>.

CREATE TABLE

In addition to the syntax given in the reference manual, some options are available.

INTERNAL	the table will become a 'SYSTEM' table and can only be specified during installation
ROWID	can be used in SQLMODE Oracle, see 'Differences to Oracle' in L1.
DYNAMIC	the table will not be balanced after delete, because it is very dynamic, that means, it is expected that one of the next inserts will fill the 'whole' produced by the delete
~~FACT~~	~~used with business warehouse to tell the SAP DB optimizer that this is the main table, the fact table~~
~~DIMENSION~~	~~used with business warehouse to tell the SAP DB optimizer that this is one of several tables 'around' the fact table~~
~~BWHIERARCHY~~	~~used with business warehouse~~

The business warehouse options are not used any more

ALTER TABLE <table name> [NOT] DYNAMIC

With this command the dynamic-option (see CREATE TABLE) is changed.

CHECK TABLE

CHECK TABLE <table name> [NO SAVEPOINT]

This statement causes the SAP DB task to check the structure of the b*tree containing the data of the table identified by <table name>.

If the structure is ok but the bad-flag of the table was set before, it is unset and a savepoint is executed.

If NO SAVEPOINT is specified, no savepoint will be executed.

CHECK TABLE <table name> WITH SHARE LOCK

This statement causes the SAP DB task to check the structure of the b*tree containing the data of the table identified by <table name>. In addition to the normal CHECK TABLE it is checked if all LONG surrogates are correct, i.e. if for all identifications for LONG column values these values exist.

To avoid problems with parallel changes in this table it it share locked.

CHECK TABLE <table name> CATALOG

This statement causes the SAP DB task to check the catalog information of the table identified by <table name>. Any errors found are repaired implicitely.

CHECK DATABASE

CHECK DATABASE :database_fill LOG :log_fill The used space (in percent) of the database and the log are returned. Each is returned as a value of datatype FIXED (2).

This statement may be used with MESSAGE TYPE DBS and PARSE/EXECUTE.

INSTALLATION ON/OFF

With INSTALLATION ON the SAP DB task is told, that the installation calls it and that tables specified with 'INTERNAL' are allowed and should be 'SYSTEM' tables.

INSTALLATION OFF closes that part of the installation.

BEGIN SAVE

The following commands are exclusively used by SAP DB LOAD respectively the Replication Manager.

BEGIN SAVE TABLE

BEGIN SAVE TABLE <table name> tells the SAP DB task, that the data pages of the table identified by <table name> will be exported by the application process. The SAP DB task returns a BD-info, which is used by the following outcopy orders.

BEGIN SAVE INDEX

BEGIN SAVE INDEX <table name> <indexno> tells the SAP DB task, that the leaf pages of the index identified by <table name> and <indexno> will be exported by the application process. The SAP DB task returns a BD-info, which is used by the following outcopy orders.

BEGIN SAVE LONG

BEGIN SAVE LONG <table name> tells the SAP DB task, that the container for short LONG columns will be exported by the application process. The SAP DB task returns a BD-info, which is used by the following outcopy orders.

BEGIN SAVE COLUMN

BEGIN SAVE COLUMN <columnno> tells the SAP DB task, that a column of datatype LONG, which is stored in a B*tree will be exported by the application process.
This command is sent together with 2 additional parts: The bdinfo part contains the bdinfo (consisting of a treeid of the base table and twice the root PNO of the base table) which was returned by the SAP DB task for BEGIN SAVE TABLE <table name>. The data part contains the file position of the base table, the long column count of that table, the column number, and the key of the row containing the LONG column.
When giving the command for the first time, the file position is of no interest for the SAP DB task. The column number has to be 0 and the key has to be the zero key.

The SAP DB task then returns a BD-info. In addition the SAP DB task returns a key part which contains the new file position, the long column count (unchanged), the column number, and the key of the row containing the long value to be extracted.

The returned BD-info is used by the following outcopy order. When the count of pages is less than the maximum possible count of pages in a packet, the end of the outcopy order is reached.

After having finished the extraction of one long value, the next long value has to be requested by sending again BEGIN SAVE COLUMN, the BD-info and the values returned in the key part on the former command. This loop has to be performed until the SAP DB task sends a return code of 100 = "Row not found" for BEGIN SAVE COLUMN.

Loading

BEGINLOAD <table name> [WITH STAMP]

This statement is exclusively used by SAP DB LOAD respectively the Replication Manager.

It tells the SAP DB task that the data pages of the table identified by <table name> will be imported into the database during a FASTLOAD for this table.

As the result the SAP DB task returns a BD-info, which will be used by the following incopy messages.

LOAD <table name> (<column list>)

This statement is exclusively used by SAP DB LOAD respectively the Replication Manager in the course of a fastload statement.

The SAP DB task returns a description of the table identified by <table name> which enables the application to build the records of the table.

SAVE

SAVE CATALOG ALL INTO <table name>
SAVE CATALOG USER INTO <table name>
SAVE CATALOG OF <table name> INTO <table name>

All SAVE statements are exclusively used by SAP DB LOAD respectively the Replication Manager to get information about tables, views, indexes, etc. The result is stored in a table which has to be specified in the statement.

RESTORE

RESTORE TABLE <table name>

This statement is used by SAP DB LOAD respectively the Replication Manager to inform the SAP DB task, that the data pages of the table identified by <table name> are to be loaded into the SAP DB task. The SAP DB task returns a BD-info for the following incopy statements.

RESTORE LONG <table name>

This statement is used by SAP DB LOAD respectively the Replication Manager to inform the SAP DB task, that the data pages of the short long columns of table <table name> are to be loaded into the SAP DB task. The SAP DB task returns a BD-info for the following incopy statements.

RESTORE COLUMN <table name>

This statement is used by SAP DB LOAD respectively the Replication Manager to inform the SAP DB task, that the data pages of a long column of table <table name> is to be loaded into the SAP DB task. The column is identified by the key and the column number. The SAP DB task returns a BD-info for the following incopy statements.

ENDLOAD <table name>

This statement is exclusively used by SAP DB LOAD respectively the Replication Manager. It tells the SAP DB task that an incopy operation of pages has been finished.

END

END LOAD <table name>

This statement is exclusively used by SAP DB LOAD respectively the Replication Manager. It tells the SAP DB task that an incopy operation of the catalog of the table has been finished.

END RESTORE

This statement is exclusively used by SAP DB LOAD respectively the Replication Manager. It tells the SAP DB task that a restore operation has been finished.

STATISTICS

LOAD STATISTICS

The statistics stored in the table SYSDBA.SYSSTATISTICS are transfered into the catalog.

UNLOAD STATISTICS

The statistics (i.e. number of rows, number of pages, distinict values) of all base tables and the corresponding indexes are stored in the table SYSDBA.SYSSTATISTICS.

OPTIMIZE ON/OFF

As default it will be tried to convert SELECTs on complex views from two internal selects into one step. With OPTIMIZE OFF this can be avoided.

This may be necessary in case of errors in the code of select conversion. With OPTIMIZE ON the default behaviour can be selected again.

DBPROCEDURE

CREATE [OR REPLACE] DBPROC[EDURE]

```
CREATE [OR REPLACE] DBPROC[EDURE] <dbproc name>
[(<formal parameter>,..)] IN <package name> EXECUTE INPROC
```

This statement is sent (usually during installation) when a procedure implemented in a programming language is to be imported into the SAP DB catalog.

This causes the SAP DB task to store the interface of the application method into the catalog.

[DB]PROC[EDURE]

[DB]PROC[EDURE] `<dbproc name>` [([<value expression>] [,...])] [WITH COMMIT]

CALL `<dbproc name>` [([<value expression>] [,...])] [WITH COMMIT]

This statement executes a DBPROCEDURE in the SAP DB task. See CALL statement in the reference manual.

DBFUNCTION

CREATE [OR REPLACE] DBFUNCTION

```
CREATE [OR REPLACE] DBFUNCTION <db function>
[(<formal parameter>,..)] IN <package name>
[USING <function spec>,...]

<function spec> ::=
    <function name> [( VARCHAR | NUMBER )]
```

This statement is sent if a dbfunction is created.

This feature is not used any more.

DROP DBFUNCTION

DROP DBFUNCTION <name> is sent by the PL or SAP DBIC workbench to remove a dbfunction from the SAP DB task.

CURRENT USER <username>

Within a ddl-trigger or as internal command this command will change the name of the current user from the name given during CONNECT or the last CURRENT USER to the username given now.

FORCE SAVEPOINT [NO OPEN TRANSACTION]

With this statement one can force a savepoint of the SAP DB task. If NO OPEN TRANSACTION is specified, not only a savepoint, but a checkpoint is forced.

RESTART INDEX

All indexes which are marked as not to be accessible, are implicitely dropped and created again. After execution of this statement all indexes should be accessible.

WRITE CACHE

This command dumps the AK-cache of this session.