NAME

gds_$attach_database −open database

SYNTAX

```
status = gds_$attach_database (
status_vector.vector_long.out,
db_name_length.ushort.in,
db_name.vector_char.in,
db_handle.ulong.inout,
parm_buffer_length.ulong.in,
parm_buffer_address.vector_byte.in)
```

DESCRIPTION

The **gds_$attach_database** routine opens an existing database for program access. Chapter 3 discusses the use of this routine.

For many applications, you may find that the **ready** statement is easier to code than a call to **gds_$attach_database**. Except for overriding the two defaults on a database attach, the statement and the attach call are functionally equivalent. However, if you want to override the defaults on the attach, you should call the **gds_$attach_database** routine.

Also, if the module does not contain any other statements, you might call the **gds_$** routine to avoid the extra step of preprocessing the program with **gpre** to handle the **ready** statement.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

db_name_length
db_name The length of the file name and the file name of the file that contains the database. If the length is zero, the access method assumes that *db_name* is null-terminated. If you program in C, you can take advantage of this convention. However, non-C programs must supply a non-zero value for the length.

db_handle Identifier for the database you want to attach. The handle must be zero at the time of the call. Otherwise, the access method returns an error.

parm_buffer_length
parm_buffer_address The length in bytes and address of the parameter buffer. The calling program uses the parameter buffer to pass information about the database to the access method. The parameter buffer consists of a version number followed by a contiguous series of *clumplet*s:

Syntax: Clumplet Format

parm_buffer ::= *version_number clumplet...*
version_number ::= **1**
clumplet ::= *type*.ubyte *length*.ubyte *value*.vector_byte

Each clumplet describes itself, including an item *type* to describe the parameter being passed, the *length* of the clumplet, and the *value* being passed.

DPB PARAMETERS

Parameters for the database parameter block fall into three catagories. They are:

- Special control on a normal attach (**gds_$dpb_dbkey_scope** and **gds_$dpb_num_buffers**).
- System management functions (**gds_$dpb_sweep**, **gds_$dpb_verify**, **gds_$dpb_enable_journal**, and **gds_$dpb_disable_journal**). Several parameters are used to invoke system management functions. Some of them require exclusive access to the database. When you call **gds_$attach_database** with a *dpb* parameter that implies exclusive access, the system will wait until all users finish before attaching the database. When the function completes, the database is attached and can be used for normal data access functions.
- Database creation. Such parameters apply only to newly created databases and are described in the manual page for **gds_$create_database**.

A description of the *dpb* parameters used in calls to the attach database routine follows:

gds_$dpb_dbkey_scope Scope of dbkey context. If you explicitly reference a dbkey, the access method returns either the same record it returned when you last referenced the dbkey or the error code **gds_$bad_dbkey**. The format of this clumplet is:

type	ubyte	**gds_$dpb_dbkey_scope**
length	ubyte	length of clumplet in bytes (1)
value	byte	0 (entire transaction) or 1 (entire database session)

gds_$dpb_num_buffers This parameter sets the number of buffers allocated for use with the database. The number must be between 10 and 100. Each buffer holds one database page, so the buffer size is determined by the database page size. The default is 25 buffers. Increasing the number of buffers will improve performance for multi-way joins. You can reduce the number of buffers when access is primarily through a single relation. The format of this clumplet is:

type	ubyte	**gds_$dpb_num_buffers**
length	ubyte	length of clumplet in bytes (1)
value	byte	number of buffers to allocate

gds_$dpb_sweep This parameter causes the access method to read all records in the database and remove

versions that are no longer needed. Old versions are always removed when the record is accessed, so an active database does not need sweeping. If, however, some records are modified intensely, then ignored, the sweep option will free unused space. The format of this clumplet is:

type	ubyte	**gds_$dpb_sweep**
length	ubyte	length of clumplet in bytes (1)
value	ubyte	gds_$dpb_records

gds_$dpb_verify This parameter causes the access method to validate that internal structures are consistent. It requires exclusive access to the database. The format of this clumplet is:

type	ubyte	**gds_$dpb_verify**
length	ubyte	length of clumplet in bytes (2)
value	uword	suboption bits. Suboption flags of **gds_$dpb_verify** direct the system to do a more or less complete validation, and to correct or only report errors it finds. The suboptions are: — **gds_$dpb_pages** (default): Verifies that all pages not in the free list are actually in use, and that the structure of every page is correct. By default, orphaned pages are returned to the free list. — **gds_$dpb_records**: Verifies that all records and record fragments are linked to relations, removes all old versions, and validates the internal structure of records. By default, space used by orphaned records and old versions is reclaimed, and the transaction inventory pages are reset. — **gds_$dpb_no_update**: Reports errors, but makes no changes to the database. Specifically, it does not add pages to the free list or reclaim space. — **gds_$dpb_repair**: Corrects errors even if the correction may involve the loss of data.

gds_$dpb_enable_journal This parameter names the journaling subsystem that will maintain an after-image journal for the database. It requires exclusive access to the database. The format of this clumplet is:

type	ubyte	**gds_$dpb_enable_journal**
length	ubyte	length of journal name in bytes
value	vector_char	journal system name

gds_$dpb_disable_journal This parameter turns off after-image journaling for the database. It requires exclusive access to the database. The format of this clumplet is:

type	ubyte	**gds_$dpb_disable_journal**
length	ubyte	0

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_17
        CHAR  *database, string [64], *p, *q, *dpb, journal [256];
        long  *handle, status_vector [20];

        /* get database and journal file names from input command */

        p = dpb = string;
        *p++ = dpb_enable_journal;
        *p++ = strlen (journal);
        for (q = journal; *q;)
            *p++ = *q++;
        dpb_length = p - dpb;

        gds_$attach_database (status_vector,
          0,  /* name is null terminated */
          *database,
          handle,
          dpb_length,
          *dpb);

        if (status_vector [1])
          gds_$print_status (status_vector);

        if (handle)
          gds_$detach_database (status_vector, handle);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$create_database**
- **gds_$database_info**
- **gds_$create_database**

See also the **ready** statement in

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$blob_info −blob information call

SYNTAX

status = **gds_$blob_info** (
status_vector.vector_long.out,
blob_handle.ulong.in,
item_list_buffer_length.ushort.in,
item_list_buffer_address.vector_byte.in,
result_buffer_length.ushort.in,
result_buffer_address.unspec.out)

DESCRIPTION

The **gds_$blob_info** routine provides information about an open blob. You can call **gds_$blob_info** to inquire about blob characteristics, such as how much space your program needs to process it.

The calling program passes its request for information through the item list buffer, and returns the information to the result buffer. See Chapter 2 for an example of a call to a similar routine (**gds_$database_info**) and the parsing of the result buffer.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

The returned status code indicates only that the access method accepted the request for information; it does not mean that it understood the request or that it supplied all requested information. Your program must interpret the contents of the result buffer.

blob_handle Identifies the blob about which you would like some information. A call to **gds_$create_blob** or **gds_$open_blob** establishes this handle.

item_list_buffer_length
item_list_buffer_address Provides the length and address of the item list buffer. The item list buffer is a regular byte vector with no structure. The calling program lists the items about which it requires information in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

result_buffer_length
result_buffer_address Provides the length and address of the result buffer. The access method returns the requested information to the result buffer. The result buffer has the following format:

Syntax: Information Call Result Buffer Clumplet

result_buffer ::= clumplet...

clumplet ::= *type*.ubyte *length*.ushort *value*.short

The value of *type* is the item you requested in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

The clumplets returned to the result buffer are not aligned. Furthermore, binary numbers are in a generic format, which you must convert to a datatype native to your computer before interpreting them. In a generic binary value, the least significant byte is first, and the most significant is last. The sign is in the last byte. To interpret a binary value returned by an information call:

- Determine the size, which can be *1*, *2*, or *4* bytes.
- Reverse the order of the bytes.

The following routine converts the contents of the result buffer into something you can read:

```
REV_integer (ptr, length)
          unsigned char *ptr;
          short length;
        {
        /************************************
         *
         *  R E V _ i n t e g e r
         *
         ************************************
         *
         * Functional description:
         *  Pick up (and convert) an integer
         *  of length 1, 2, or 4 bytes.
         *
         ************************************/
        int     value;
        short   shift;

        value = shift = 0;

        while (--length >= 0)
          {
          value += (*ptr++) << shift;
          shift += 8;
          }

        return value;
        }
```

INFORMATION ITEMS

You can ask about the following items in the item list buffer:

gds_$info_blob_num_segments The number of segments that comprise the blob field. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_blob_num_segments**
length	ushort	length of clumplet in bytes
value	unspec	total number of blob segments

gds_$info_blob_max_segment The length of the longest segment in the blob field. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_blob_max_segment**
length	ushort	length of clumplet in bytes
value	unspec	length of longest segment

gds_$info_blob_total_length The total length of the blob. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_blob_total_length**
length	ushort	length of clumplet in bytes
value	unspec	total length of blob field

gds_$info_blob_type The blob type. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_blob_type**
length	ushort	length of clumplet in bytes
value	unspec	0 (segment) or 1 (reserved)

In addition to the above items for which you can request information, the access method may also return the following status messages to the result buffer:

gds_$info_end End of result buffer with no errors. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_end**
length	ushort	length of clumplet (0)

gds_$info_truncated Input into the result buffer was truncated. The access method returns a truncated clumplet as the last clumplet in the result buffer if the result buffer was not large enough to hold all the information you requested. If your program encounters this clumplet, it means that all preceding information is valid, but at least one item is missing. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_truncated**

gds_$info_error An error. The access method returns an error clumplet if an item of requested information was not available. This clumplet has the same form as other clumplets, but the information portion contains only the information type value and a code indicating why the information was not available. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_error**
length	ushort	length of clumplet (2)
value	short	rude error message

EXAMPLE

```
 static char  blob_items [] = {
             gds_$info_max_segment,
             gds_$info_number_segments,
             gds_$info_blob_type};


         CHAR  blob_info [32];

         /* Open the blob and get its vital statistics */

         if (gds_$open_blob (status_vector,
             DB,
             gds_$trans,
             blob,
             *blob_id))
           error ("gds_$open_blob failed", status_vector);

         if (gds_$blob_info (status_vector,
             blob,
             sizeof (blob_items),
             blob_items,
             sizeof (blob_info),
             blob_info))
           error ("gds_$blob_info failed", status_vector);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$create_blob**
- **gds_$open_blob**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$cancel_blob −remove blob

SYNOPSIS

status = **gds_$cancel_blob** (
status_vector.vector_long.out,
blob_handle.ulong.inout)

DESCRIPTION

The **gds_$cancel_blob** statement releases internal storage used by a discarded blob and sets the blob handle to null.

When you create a blob, temporarily stores it in the database. If you fail to close the blob, the temporary storage space remains allocated. Furthermore, the handle is not null, ready to cause problems for anything so unwise as to trip over it.

Because a call to this routine does *not* produce an error if the handle is null, it is good practice to call this routine before you call either **gds_$open_blob** or **gds_$create_blob**. This practice ensures that the access method cleans up earlier blob operations. If you abort a blob operation or if you do not trust the routine that passed the blob id, you should call **gds_$cancel_blob** before opening or creating a blob.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

blob_handle Identifies the blob you want to cancel. This routine sets the handle to zero. Unlike other **gds** routines, this routine returns success even if the handle is null.

EXAMPLE

```
. source:
        gds_$cancel_blob (*gds_null, blob);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$close_blob**
- **gds_$open_blob**
- **gds_$create_blob**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$close_blob −finish blob

SYNOPSIS

status = **gds_$close_blob (**
status_vector.vector_long.out,
blob_handle.ulong.inout)

DESCRIPTION

The **gds_$close_blob** statement releases system resources associated with blob update or retrieval. You should call **gds_$close_blob** as soon as you finish reading or writing a blob.

If you fail to close a blob you created, you may lose some of the data. Because the remote interface buffers segment transfer between participating nodes, it may truncate the last segment you write unless you explicitly close the blob.

You cannot read from or write to a closed blob without re-opening it with a call to **gds_$open_blob**.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

blob_handle A non-zero value established in a call to **gds_$create_blob** or **gds_$open_blob** that identifies the blob you want to close. This routine sets the value of the handle to zero.

EXAMPLE

```
if (gds_$close_blob (status_vector,
          from_blob))
          ERRQ_database_error (from_dbb, status_vector);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$cancel_blob**
- **gds_$open_blob**
- **gds_$create_blob**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$commit_transaction −commit transaction

SYNOPSIS

status = **gds_$commit_transaction** (
status_vector.vector_long.out,
transaction_handle.ulong.inout)

DESCRIPTION

The **gds_$commit_transaction** routine commits an active transaction. A successful call to **gds_$commit_transaction**:

- Certifies database changes made during the transaction as permanent
- Unwinds active requests
- Cancels open blobs

If you have questions about the importance or effects of the commit operation, you are reading the wrong book. See Chapter 4 of this manual for a discussion of transactions.

The access method automatically executes a call to **gds_$prepare_transaction** for transactions that update more than one database. However, you can call the prepare routine yourself. See Chapter 4 for more information about the two-phase commit operation.

If you are writing an interactive utility that starts transactions automatically for the user, you should put an automatic commit and detach in the normal exit routine to avoid the confusion that results when an inexperienced user undoes a morning's work by exiting and rolling back.

Note that a **commit** statement is shorter and easier to code than a call to **gds_$commit_transaction**. However, if the program or module does not contain any other statements, you might call the **gds** routine to avoid the extra step of preprocessing the program with **gpre** to handle the **commit** statement.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

transaction_handle A non-zero value that identifies the transaction you want to commit. A call to **gds_$start_transaction** establishes this handle. If the call to this routine is successful, the access method sets the transaction handle to zero. Otherwise, it leaves it unchanged.

EXAMPLE

```
. double backslash n in example
        if (gds_$commit_transaction (status, trans))
          {
          fprintf ('Battle stations, battle stations!\n');
          gds_$print_status (status);
          }
```

SEE ALSO

See the entries in this chapter for:

- **gds_$prepare_transaction**
- **gds_$unwind_request**
- **gds_$cancel_blob**
- **gds_$transaction_info**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$compile_request −compile request

SYNOPSIS

status = **gds_$compile_request** (
status_vector.vector_long.out,
db_handle.ulong.in,
request_handle.ulong.inout,
blr_string_length.ushort.in,
blr_string_address.vector_byte.in)

DESCRIPTION

The **gds_$compile_request** routine compiles a request passed in BLR form into an internal format that the access method can execute. The internal format, called an *execution tree*, has all field and relation references resolved, all view and computed field references expanded, and its access strategy optimized.

The compile request call does not take a transaction handle, so compiled requests are not bound to any one transaction. Therefore, you can start a request that was compiled during a late, perhaps lamented transaction. Call **gds_$start_request** and pass the request handle returned by the call to **gds_$compile_request** to start a request.

By saving a compiled request for re-use, you avoid the cost of a second compilation. In general, saving compiled requests is good idea *if they will be used again*. An interactive program that generates requests to satisfy user queries should probably not save compiled requests because it is unlikely that it can re-use a request or even match a compiled request to a new query,

Because a compiled request occupies memory, release it if you know that it will not be re-used. Call **gds_$release_request** to release memory and other system resources associated with a compiled request.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

dbhandle Identifier for the database against which the request will be executed. A call to **gds_$attach_database** establishes this handle.

request_handle Identifier returned by the access method. The handle identifies the request you want compiled for calls to **gds_$start_request**, **gds_$start_and_send**, **gds_$request_info**, and **gds_$unwind_request**. The request handle must be null on input.

blr_string_length
blr_string_address The length and address of the BLR string that contains the request.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_compile
        . double backslash n in example
        request = NULL;
        blr_length = blr - blr_buffer;
```

```
if (gds_$compile_request (status_vector,
    GDS_REF (DB),
    GDS_REF (request),
    blr_length,
    blr_buffer))
  {
  ERROR ("gds_$compile failed\n", status_vector);
  }
```

SEE ALSO

See the entries in this chapter for:

- **gds_$start_request**
- **gds_$release_request**

DIAGNOSTICS

The access method returns an error if:

- The BLR string contains values that are not defined, incorrect combinations of values, or references to objects that do not exist in the database.
- The metadata you pass is invalid. You may encounter this problem if you have changed the metadata since you created the BLR.

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$create_blob −store new blob

SYNOPSIS

status = **gds_$create_blob** (
status_vector.vector_long.out,
db_handle.ulong.in,
transaction_handle.ulong.in,
blob_handle.ulong.inout,
blob_id.uquad.out)

DESCRIPTION

The **gds_$create_blob** statement creates the context for storing a blob and opens the blob for write access.

A successful call to **gds_$create_blob** creates the environment for storing a blob. However, the access method does not store the blob until a **blr_assignment** statement assigns it to a relation. Chapter 7 provides a detailed description of the steps involved in storing a blob.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

db_handle Identifier for the database where the blob will be created. A call to **gds_$attach_database** establishes this handle.

transaction_handle Identifier for the transaction in which the blob will be created. A call to **gds_$start_transaction** establishes this handle.

blob_handle Identifier returned by the access method. The blob handle is a name that identifies the new blob in the context of the current transaction. The blob handle must be zero on input.

blob_id Internal identifier for the blob assigned by the access method. The identifier must have a value of *gds_$blob_null*.

The access method uses *blob_id* when it opens the blob with a call to **gds_$open_blob**. However, the value of *blob_id* when you create the blob is *not* the same as when you open the blob.

When you create the blob, it is essentially an ‘‘orphan’’ until a **blr_assignment** statement assigns the value of *blob_id* to the blob field in a relation.

The access method automatically changes the value of *blob_id* at the time of assignment. Once you assign *blob_id* to its relation, the creation value disappears forever. Therefore, if you open the newly created blob later, you must read *blob_id* from the record. If you try to save the old *blob_id* and re-use it, the access method returns an error.

EXAMPLE

```
to_blob = NULL;
        if (gds_$create_blob (status_vector,
            to_dbb_handle,
            to_dbb_transaction,
            to_blob,
            *to_dsc_address))
          ERRQ_database_error (to_dbb_handle, status_vector);
```

SEE ALSO

See the entry in this chapter for:

- **gds_$put_segment**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$create_database −create new database

SYNOPSIS

```
status = gds_$create_database (
status_vector.vector_long.out,
db_name_length.ushort.in,
db_name.vector_char.in,
db_handle.ulong.inout,
parm_buffer_length.ulong.in,
parm_buffer_address.vector_byte.in)
```

DESCRIPTION

The **gds_$create_database** routine creates a new, empty database, and attaches it for the calling program. Although the database contains no user data, it does contain a full set of system relations that describe themselves.

In general, you rarely need this routine. The primary users of the **gds_$create_database** routine are **gdef** and the restore process of **gbak**. You will call this routine if you develop an application that dynamically creates individual databases for an electronic mail system, calendar management, a digital bulletin board, and so on. Another use would be a data definition utility intended for a particular environment.

This routine supersedes any database with the same name. Therefore, if you want to keep your present databases, try to attach a database before you create it. If the attach succeeds, do not call create.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

db_name_length
db_name The length of the file name and the file name of the file that will contain the database. If the length is zero, the access method assumes that *db_name* is null-terminated. If you are programming in C, you can take advantage of this convention. Non-C programmers must supply a non-zero value for the length.

db_handle Identifier for the database you want to attach. The handle must be zero at the time of the call. Otherwise, the access method returns an error.

parm_buffer_length
parm_buffer_address The length in bytes and address of the parameter buffer. The calling program uses the parameter buffer to pass information about the database to the access method. The parameter buffer consists of a version number followed by a contiguous series of *clumplets*:

parm_buffer ::= *version_number clumplet...*

version_number ::= **1**

clumplet ::= *type*.byte *length*.ubyte *value*.vector_byte

Each clumplet describes itself, including a *type*, the *length* of the clumplet, and the *value* you want to set.

DPB PARAMETERS

You can pass the following values to the access method:

gds_$dpb_page_size Page size in blocks. The default page size is 1024 bytes, but you should override the default if you have a very large database. The choices for page size are 1024, 2048, 4096, and 8192. If your database will contain relations with more than 20,000 records, you should increase the page size from the default to 2048. For much larger relations, increase the size again. The purpose of increasing the page size is to increase the size of an index page and reduce the depth of the index tree. Because of key compression, there is no hard and fast rule on the size of an index entry. An index node has four bytes of overhead, plus the compressed key. There are ten bytes of overhead on the page.

Note that you can write a simple program to create a database with a larger page size, and then use **gdef** to modify that database to include your relations, fields, and so on.

This clumplet has the following format:

type	ubyte	*gds_$dpb_page_size*
length	ubyte	length of clumplet
value	byte	page size in bytes

gds_$dpb_num_buffers This parameter sets the number of buffers allocated for use with the database. The number must be between 10 and 100. Each buffer holds one database page, so the buffer size is determined by the database page size. The default is 25 buffers. Increasing the number of buffers will improve performance for multi-way joins. You can reduce the number of buffers when access is primarily through a single relation. The format of this clumplet is:

type	ubyte	*gds_$dpb_num_buffers*
length	ubyte	length of clumplet in bytes (1)
value	byte	number of buffers to allocate

gds_$dpb_dbkey_scope Scope of dbkey context. If you explicitly reference a dbkey, the access method returns either the same record it returned when you last referenced the dbkey or the error code *gds_$bad_dbkey*. The format of this clumplet is:

type	ubyte	*gds_$dpb_dbkey_scope*
length	ubyte	length of clumplet in bytes (1)
value	byte	0 or 1

The scope can be the entire transaction (value of *0*) or the entire database attach session (value of *1*).

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_18
        if (gds_$create_database (gds_$status,
            0,  /* name is null-terminated */
            *file_name),
            DB,
            0, 0)) /* use default page size */
          {
          gds_$print_status (gds_$status);
          sprintf (s, "Couldn't create database \"%s\"", file_name);
          }
```

SEE ALSO

See the entry in this chapter for:

- **gds_$attach_database**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$database_info −database information call

SYNTAX

```
status = gds_$database_info (
status_vector.vector_long.out,
db_handle.ulong.in,
item_list_buffer_length.ushort.in,
item_list_buffer_address.vector_byte.in,
result_buffer_length.ushort.in,
result_buffer_address.unspec.out)
```

DESCRIPTION

The **gds_$database_info** routine returns information about an attached database. You might call **gds_$database_info** for the following reasons:

- Prepare to reconnect to transactions in limbo. This is, in fact, the primary use for calling the **gds_$database_info** routine. If a transaction fails after a successful call to **gds_$prepare_transaction**, but before a call to **gds_$commit_transaction** completes, that transaction becomes a ‘‘zombie’’ and must be forcibly terminated. The **gfix** utility locates and eliminates zombies, but you may need more direct control in a sophisticated, critical application.

  See Chapter 2 for an extract from **gfix**, a utility that performs such functions.
- Determine how much space is used for page caches. The space is, of course, the product of the number of buffers and the page size. To affect this information, you must first attach the database, call **gds_$database_info**, then detach and attach again using the database parameter block.
- Monitor performance. For example, you might want to compare the efficiency of two update strategies, such as updating a sorted or unsorted stream.

The calling program passes its request for information through the item list buffer, and the access method returns the information to the result buffer. See Chapter 2 for an example of a call to **gds_$database_info** and the parsing of the result buffer.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

The returned status code indicates only that the access method accepted the request for information; it does not mean that it understood the request or that it supplied all requested information. Your program must interpret the contents of the result buffer.

db_handle Identifies the database about which you want information. A call to **gds_$attach_database** establishes this handle.

item_list_buffer_length
item_list_buffer_address Provides the length and address of the item list buffer. The item list buffer is a

regular byte vector with no structure. The calling program lists the items about which it requires information in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

result_buffer_length
result_buffer_address Provides the length and address of the result buffer. The access method returns the requested information to the result buffer. The result buffer has the following format:

Syntax: Information Call Result Buffer Clumplet

result_buffer ::= clumplet...
clumplet ::= *type*.ubyte *length*.ushort *value*.short

The value of *type* is the item you requested in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

The clumplets returned to the result buffer are not aligned. Furthermore, binary numbers are in a generic format, which you must convert to a datatype native to your computer before interpreting them. In a generic binary value, the least significant byte is first, and the most significant is last. The sign is in the last byte. To interpret a binary value returned by an information call:

- Determine the size, which can be *1*, *2*, or *4* bytes.
- Reverse the order of the bytes.

The following routine converts the contents of the result buffer into something you can read:

```
REV_integer (ptr, length)
          unsigned char *ptr;
          short length;
        {
        /************************************
         *
         *  R E V _ i n t e g e r
         *
         ************************************
         *
         * Functional description:
         *  Pick up (and convert) an integer
         *  of length 1, 2, or 4 bytes.
         *
         ************************************/
        int     value;
        short   shift;

        value = shift = 0;

        while (--length >= 0)
          {
```

```
  value += (*ptr++) << shift;
  shift += 8;
  }

return value;
}
```

INFORMATION ITEMS

You can ask about the following items in the item list buffer:

gds_$info_page_size Page size of database. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_page_size**
length	short	length of clumplet in bytes
value	unspec	page size in bytes

gds_$info_num_buffers Number of buffers currently allocated. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_num_buffers**
length	short	length of clumplet in bytes
value	unspec	number of allocated buffers

gds_$info_limbo Identification numbers of transactions in limbo. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_limbo**
length	short	length of clumplet
value	longword *	vector of transaction ids

In addition to the above items for which you can request information, the access method may also return the following status messages to the result buffer:

gds_$info_end End of result buffer with no errors. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_end**
length	short	length of clumplet (0)

gds_$info_truncated Input into the result buffer was truncated. The access method returns a truncated clumplet as the last clumplet in the result buffer if the result buffer was not large enough to hold all the information you requested. If your program encounters this clumplet, it means that all preceding information is valid, but at least one item is missing. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_truncated**

gds_$info_error Error. The access method returns an error clumplet if an item of requested information was not available. This clumplet has the same form as other clumplets, but the information portion contains only the information type value and a code indicating why the information was not available. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_error**
length	short	length of clumplet
value	short	rude error message

The **gds_$database_info** routine can return information about items that mean more to the developers of the access method than they will to your application. However, you can ask about them if you really want to know:

gds_$info_reads Number of page reads since the database was attached. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_reads**
length	short	length of clumplet
value	longword	number of page reads since last attach call

gds_$info_writes Number of page writes since the database was attached. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_writes**
length	short	length of clumplet
value	longword	number of page writes since last attach call

gds_$info_fetches Number of internal page accesses since the database was attached. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_fetches**
length	short	length of clumplet
value	longword	number of page accesses since last attach call

gds_$info_marks Number of internal page update declarations since the database was attached. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_marks**
length	short	length of clumplet
value	longword	number of page update declarations since last attach call

gds_$info_max_memory Most memory used at one time since the database was attached. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_max_memory**
length	short	length of clumplet
value	longword	highwater point of memory usage since last attach call

gds_$info_current_memory Amount of memory currently in use. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_current_memory**
length	short	length of clumplet
value	longword	amount of memory currently in use

EXAMPLE

The following call finds out about transactions in limbo for subsequent reconnects or rollbacks:

```
if (gds_$database_info (status_vector,
            handle,
            sizeof (limbo_info),
            limbo_info,
            sizeof (buffer),
            buffer))
          {
```

```
gds_$print_status (status_vector);
return;
```

} You must interpret the contents of the result buffer. The **gds_$print_status** routine displays the contents of the status vector. See Chapter 2 for an example of code that parses the information returned to the result buffer.

SEE ALSO

See the entry in this chapter for:

- **gds_$attach_database**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$detach_database −detach and close database

SYNTAX

status = **gds_$detach_database** (
status_vector.vector_long.out,
db_handle.ulong.inout)

DESCRIPTION

The **gds_$detach_database** routine detaches an attached database. You should call this routine (or issue a **finish** statement) to release system resources when you are finished using a database. Detaching a database reduces the use of virtual memory by releasing:

- The cache
- Mapping windows
- Data structures such as compiled requests and transaction state information

If your program is attached to a remote database, a call to **gds_$detach_database** also releases the buffers and structures that control the remote interface on your node and the remote server on the node where the database is stored.

If you call **gds_$detach_database** while there are active transactions, the access method rolls back those transactions. If there are any transactions in limbo, they stay there.

For most applications, the **finish** statement is easier to code than a call to **gds_$detach_database**. Both perform the same functions. There are no options that you can choose on the detach call.

However, you can call **gds_$detach_database** if your program does not contain any other statements. Coding the detach call eliminates the need for preprocessing the program with **gpre**.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

db_handle Identifier for the database you want to detach. A call to **gds_$attach_database** or **gds_$create_database** establishes this handle. A successful return from the detach call sets the handle to null. The database handle must not be null on input.

EXAMPLE

The following statement detaches a database:

```
if (handle)
          gds_$detach_database (status_vector, handle);
```

See Chapter 2 for an example of this program in context.

SEE ALSO

See the entries for **gds_$attach_database** and **gds_$create_database** in this chapter.

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$get_segment −read segment

SYNOPSIS

status = **gds_$get_segment** (
status_vector.vector_long.out,
blob_handle.ushort.in,
actual_segment_length.ushort.out,
segment_buffer_length.ushort.in,
segment_buffer_address.unspec.out)

DESCRIPTION

The **gds_$get_segment** routine reads a portion of a blob field. This routine is the *read* call for blob manipulation. Before you can read a blob, you must open it with a call to **gds_$open_blob** or an equivalent routine. You may want to handle blobs with direct calls even in primarily and especially in programs.

What a call to **gds_$get_segment** does depends on previous blob calls. If the last call that used *blob_handle* was:

- **gds_$get_blob**, it reads the next segment.
- **gds_$open_blob**, it reads the first segment.

Chapter 7 provides a detailed description of the steps involved in accessing a blob.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

blob_handle Identifier for the blob. A call to **gds_$open_blob** during the current transaction establishes this handle.

actual_segment_length Number of bytes actually passed to the segment buffer by the access method. This parameter is useful if the blob segment is shorter than the segment buffer.

If the blob segment is longer than the buffer, the blob segment will be truncated. In this case, the access method returns the status code *gds_$segment* to indicate that the segment buffer contains a truncated segment.

segment_buffer_length
segment_buffer_address The length and address of the segment buffer into which the access method reads blob segments.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_get_segment
       /*
        * Copy one blob into another.  First create the new blob,
```

```
 * then open the old one, and loop getting segments from
 * one and putting them to the other.  Finally, close both.
 * ERRQ_database_error prints the database status and performs
 * reasonable cleanup.
 */

if (gds_$create_blob (status_vector,
    to_dbb_handle,
    to_dbb_transaction,
    to_blob,
    *to_dsc_address))
  ERRQ_database_error (to_dbb_handle, status_vector);

if (gds_$open_blob (status_vector,
    from_dbb_handle,
    from_dbb_transaction,
    from_blob,
    *from_dsc_address))
  ERRQ_database_error (from_dbb_handle, status_vector);

while (!gds_$get_segment (status_vector,
    from_blob,
    length,
    sizeof (buffer),
    buffer))
    if (gds_$put_segment (status_vector,
    to_blob,
    length,
    buffer))
    ERRQ_database_error (to_dbb, status_vector);

if (gds_$close_blob (status_vector,
        from_blob))
  ERRQ_database_error (from_dbb, status_vector);

if (gds_$close_blob (status_vector,
    to_blob))
  ERRQ_database_error (to_dbb, status_vector);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$get_blob**
- **gds_$open_blob**
- **gds_$blob_info**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$open_blob −prepare a blob for retrieval

SYNOPSIS

status = **gds_$open_blob** (
status_vector.vector_long.out,
db_handle.ulong.in,
transaction_handle.ulong.in,
blob_handle.ulong.inout,
blob_id.uquad.in)

DESCRIPTION

The **gds_$open_blob** routine opens a blob so that its data may be retrieved.

It makes good sense to call **gds_$cancel_blob** before you open a blob.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

db_handle Identifier for the database that contains the blob. A call to **gds_$attach_database** establishes this handle.

transaction_handle Identifier for the transaction in which the blob will be processed. A call to **gds_$start_transaction** establishes this handle.

blob_handle Identifier returned by the access method. The blob handle uniquely identifies the blob that you want opened. Read this from the field in your target record.

blob_id Internal identifier of the blob assigned by the access method. The identifier must be zero on input.

EXAMPLE

```
if (gds_$open_blob (status_vector,
            from_dbb_handle,
            from_dbb_transaction,
            from_blob,
            *from_dsc_address))
          ERRQ_database_error (from_dbb_handle, status_vector);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$cancel_blob**
- **gds_$close_blob**
- **gds_$get_segment**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$prepare_transaction −prepare to commit

SYNOPSIS

status = **gds_$prepare_transaction** (
status_vector.vector_long.out,
transaction_handle.ulong.in)

DESCRIPTION

The **gds_$prepare_transaction** routine performs the first phase of a two-phase commit for transactions that involve more than one database. If you do not call **gds_$prepare_transaction** before trying to commit such a transaction the access method automatically calls it when you call **gds_$commit_transaction**.

You may also want to call **gds_$prepare_transaction** if you have to coordinate database activity with non-database operations.

Chapter 4 discusses the use of this routine for both single and multiple database transactions.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

transaction_handle Identifies the transaction to commit. A call to one of the start transaction routines establishes this handle.

EXAMPLE

```
. double backslash n in example
        if (gds_$prepare_transaction (status_vector, trans)
          {
          fprintf ("\n*** error during prepare ***\n");
          gds_$print_status (status_vector);
          locate_and_fix ("rollback");
          }
```

SEE ALSO

See the entries in this chapter for:

- **gds_$commit_transaction**
- **gds_$rollback_transaction**
- **gds_$transaction_info**

See also the discussion of the **gbak** utility in

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$put_segment −write blob

SYNOPSIS

status = **gds_$put_segment** (
status_vector.vector_long.out,
blob_handle.ulong.in,
segment_buffer_length.ushort.in,
segment_buffer_address.unspec.in)

DESCRIPTION

The **gds_$put_segment** routine writes the next segment or portion of a blob.

You cannot read segments written with calls to **gds_$put_segment** until you close the blob with a call to **gds_$close_blob**.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

blob_handle Identifier for the blob to which you want to write. A call to **gds_$create_blob** establishes this handle.

segment_buffer_length
segment_buffer_address The length and address of the segment buffer that passes data to the access method.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_put_blob
        if (gds_$put_segment (status_vector,
            to_blob,
            length,
            buffer))
          ERRQ_database_error (to_dbb, status_vector);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$create_blob**
- **gds_$close_blob**
- **gds_$blob_info**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$receive −receive message

SYNOPSIS

status = **gds_$receive** (
status_vector.vector_long.out,
request_handle.ulong.in,
message_type.ushort.in,
message_length.ushort.in,
message_address.unspec.out,
instantiation.ushort.in)

DESCRIPTION

The **gds_$receive** routine transfers a formatted message from the access method to the calling program.

The **gds_$receive** call accepts data sent by the access method to the program with a **blr_send** statement that processes the same message type. If there is no corresponding **blr_send** statement in the request, the access method returns a synchronization error and unwinds the request.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

request_handle Identifier for the request that defined the message. A call to **gds_$compile_request** establishes this handle, and a call to **gds_$start_request** or **gds_$start_and_send** activates the request.

message_type Flag that identifies type of message in message buffer. The **blr_message** statement provides the message type and structure for the message.

message_length
message_address Length in bytes and address of the calling program's message buffer. The **blr_message** statement provides the message type and structure for the message. Your program must provide a corresponding buffer with that structure.

instantiation Incarnation of the request. This parameter avoids the cost of recompilation in a request that is called recursively. The instantiation parameter in this call identifies the instance of request you want to hear from. See the entries for **gds_$start_request** or **gds_$start_and_send** in this chapter.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_receive
        EXEC_receive (message)
            MSG    message;
        {
        /************************************
         *
         *  E X E C _ r e c e i v e
```

```
 *
 ************************************
 *
 * Functional description
 *  Receive a message from a running request.
 *
 ************************************/
REQ  request;
long  status_vector [20];

request = message->msg_request;

if (gds_$receive (status_vector,
    request->req_handle,
    message->msg_number,
    message->msg_length,
    *message->msg_buffer,
    message->level))
  db_error (request, status_vector);
}
```

SEE ALSO

See the entries in this chapter for:

- **gds_$send**
- **gds_$start_request**
- **gds_$start_and_send**

See also the entries in Chapter 10 for:

- **blr_send**
- **blr_message**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$reconnect_transaction −reconnect transaction

SYNOPSIS

status = **gds_$reconnect_transaction** (
status_vector.vector_long.out,
db_handle.ulong.in,
transaction_handle.ulong.inout,
trans_id_length.ushort.in,
trans_id.vector_byte.in)

DESCRIPTION

The **gds_$reconnect_transaction** routine connects a transaction with a status of limbo to a new parent process. Chapter 4 discusses the conditions under which a transaction might become disconnected and how to deal with it.

When you reconnect to a transaction, the only valid operations are commit and rollback. If you try to use the reconnected transaction in any other way, the call or statement fails and the access method returns an error.

For some applications, you may choose to exert close control and write your own routines for locating and expunging transactions in limbo. The **gds_$reconnect_transaction** call supports such a utility.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

db_handle Identifier for the database against which the transaction is running. A call to **gds_$attach_database** establishes this handle.

transaction_handle Identifier for the transaction that you want to reconnect. This handle must be zero on input.

trans_id_length
trans_id Length of transaction identifier and id itself.

EXAMPLE

```
static reconnect (handle, number, switches)
          int  *handle;
          long  number;
          SHORT  switches;
        {
        /************************************
         *
         *  r e c o n n e c t
         *
         ************************************
```

```
 *
 * Functional description:
 *  Commit or rollback a named transaction.
 *  Invert_integer takes a byte length and
 *  the address of a number and reverses the
 *  bytes.
 *
 ************************************/
int  *transaction;
long  id;
long  status_vector [20];

id = INVERT_integer (&number, 4);
transaction = NULL;

if (gds_$reconnect_transaction (status_vector,
    handle,
    transaction,
    sizeof (id),
    id))
  {
  gds_$print_status (status_vector);
  return;
  }

if (switches & sw_commit)
  gds_$commit_transaction (status_vector, transaction);
else
  gds_$rollback_transaction (status_vector, transaction);

if (status_vector [1])
  gds_$print_status (status_vector);
}
```

SEE ALSO

See the entries in this chapter for:

- **gds_$prepare_transaction**
- **gds_$commit_transaction**
- **gds_$rollback_transaction**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$release_request −terminate request

SYNOPSIS

status = **gds_$release_request** (
status_vector.vector_long.out,
request_handle.ulong.inout)

DESCRIPTION

The **gds_$release_request** routine frees the memory used by the execution tree of a compiled request and sets the request handle to null. Releasing an active request causes it, and all its instantiations, to unwind.

Most compiled programs use a relatively small (fewer than 50) number of requests and re-use them as the program iterates through its internal loops. For such a program, the cost of recompilation overrides the savings of released memory.

However, for an interactive program that generates requests in response to user input, the likelihood of re-using an old request is low while the number of requests is potentially high. Such a program should call **gds_$release_request** once a request has been terminated.

You can call **gds_$release_request** from a compiled program if it makes a large (greater than 75) number of requests and you can predict that certain requests will not be re-executed.

Once you have released a request, you must compile it again before you can re-use it. Call **gds_$compile_request** to compile the request.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

request_handle Identifier for the request you want to release. A call to **gds_$compile_request** establishes this handle.

EXAMPLE

```
. /gds/harrison/work/call_int/examp_release
        GEN_release ()
        {
        /************************************
         *
         *  G E N _ r e l e a s e
         *
         ************************************
         *
         * Functional description:
         *  Release any compiled requests following
         *  execution or abandonment of a request.
         *  Just recurse around releasing requests.
         *
```

```
 ***********************************/
long  status_vector [20];

for (request = top_request; request; request = request->req_next)
  if (request->req_handle && request->req_active)
    if (gds_$release_request (status_vector,
      request->req_handle))
    gds_$print_status (status_vector);

}
```

SEE ALSO

See the entries in this chapter for:

- **gds_$unwind_request**
- **gds_$compile_request**
- **gds_$request_info**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$request_info −information call

SYNTAX

```
status = gds_$request_info (
status_vector.vector_long.out,
request_handle.ulong.in,
item_list_length.ushort.in,
item_list_buffer_address.vector_byte.in,
result_buffer_length.ushort.in,
result_buffer_address.unspec.out,
instantiation.ushort.in)
```

DESCRIPTION

The **gds_$request_info** routine returns information about a compiled request.

In all likelihood, nothing but the standard remote interface would ever use this call. The remote interface calls this routine when it sets up buffers, handles error conditions, and manages pipelined communication.

Some special applications may require a customized remote interace or a layer on top of the database to emulate multiple database requests. The **gds_$request_info** routine is essential for such applications.

The calling program passes its request for information through the item list buffer, and the access method returns the information to the result buffer. See Chapter 2 for an example of a call to a similar routine (**gds_$database_info**) and the parsing of the result buffer.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

The returned status code indicates only that the access method accepted the request for information; it does not mean that it understood the request or that it supplied all requested information. Your program must interpret the contents of the result buffer.

request_handle Identifier for the request about which you want information. A call to **gds_$compile_request** establishes this handle, and a call to **gds_$start_request** activates the request.

item_list_buffer_length
item_list_buffer_address Provides the length and address of the item list buffer. The item list buffer is a regular byte vector with no structure. The calling program lists the items about which it requires information in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

result_buffer_length
result_buffer_address Provides the length and address of the result buffer. The access method returns the requested information to the result buffer. The result buffer has the following format:

Syntax: Information Call Result Buffer Clumplet

result_buffer ::= clumplet...

clumplet ::= *type*.ubyte *length*.ushort *value*.short

The value of *type* is the item you requested in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

The clumplets returned to the result buffer are not aligned. Furthermore, binary numbers are in a generic format, which you must convert to a datatype native to your computer before interpreting them. In a generic binary value, the least significant byte is first, and the most significant is last. The sign is in the last byte. To interpret a binary value returned by an information call:

- Determine the size, which can be *1*, *2*, or *4* bytes.
- Reverse the order of the bytes.

The following routine converts the contents of the result buffer into something you can read:

```
REV_integer (ptr, length)
          unsigned char *ptr;
          short length;
        {
        /************************************
         *
         *  R E V _ i n t e g e r
         *
         ************************************
         *
         * Functional description:
         *  Pick up (and convert) an integer
         *  of length 1, 2, or 4 bytes.
         *
         ************************************/
        int     value;
        short   shift;

        value = shift = 0;

        while (--length >= 0)
          {
          value += (*ptr++) << shift;
          shift += 8;
          }

        return value;
        }
```

instantiation Incarnation of the request. A recursive routine may actually involve several active versions of the same request. This parameter specifies which instance you would like to know about.

INFORMATION ITEMS

You can ask about the following items in the item list buffer:

gds_$info_number_messages Number of different request message types. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_number_messages**
length	ushort	length of clumplet in bytes
value	unspec	number of message types referenced in compiled request

gds_$info_max_message Highest numbered message type. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_max_message**
length	ushort	length of clumplet in bytes
value	unspec	number of highest message type

gds_$info_max_send Length of longest send message. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_max_send**
length	ushort	length of clumplet in bytes
value	unspec	length of longest message to be sent

gds_$info_max_receive Length of longest receive message. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_max_receive**
length	ushort	length of clumplet in bytes
value	unspec	length of longest message to receive

gds_$info_state Current status of the request. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_state**
length	ushort	length of clumplet in bytes
value	unspec	status of request. The status of the request can be: — **gds_$info_active**: The request is busy, happy, and quiet. — **gds_$info_inactive**: Request is not running. — **gds_$info_send**: Request is waiting for program to receive a message. — **gds_$info_receive**: Request is waiting for program to send a message. — **gds_$info_select**: Request is waiting for one of several messages to be sent from program.

gds_$info_message_number Current message for send. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_message_number**
length	ushort	length of clumplet in bytes
value	unspec	what message type involved in send or receive

gds_$info_message_size Send message length. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_message_size**
length	ushort	length of clumplet in bytes
value	unspec	length of message to be sent or returned

In addition to the above items for which you can request information, the access method may also return the following status messages to the result buffer:

gds_$info_end End of result buffer with no errors. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_end**
length	ushort	length of clumplet in bytes

gds_$info_truncated Input into the result buffer was truncated. The access method returns a truncated clumplet as the last clumplet in the result buffer if the result buffer was not large enough to hold all the information you requested. If your program encounters this clumplet, it means that all preceding

information is valid, but at least one item is missing. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_truncated**

gds_$info_error An error. The access method returns an error clumplet if an item of requested information was not available. This clumplet has the same form as other clumplets, but the information portion contains only the information type value and a code indicating why the information was not available. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_error**
length	ushort	length of clumplet in bytes
value	short	rude error message

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_req_info
        static CHAR  request_info [] =
          { inf_state, inf_message_number, inf_end };

        gds_$request_info (status_vector,
          request->rrq_handle,
          current_instance,
          sizeof (request_info),
          request_info,
          sizeof (info_buffer),
          info_buffer);
```

SEE ALSO

See the entry in this chapter for:

- **gds_$compile_request**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$rollback_transaction −undoing transaction

SYNOPSIS

status = **gds_$rollback_transaction** (
status_vector.vector_long.out,
transaction_handle.ulong.inout)

DESCRIPTION

The **gds_$rollback_transaction** routine undoes changes made during a transaction.

A successful call to **gds_$rollback_transaction** closes requests and blobs. A call to this routine can fail only if:

- You pass an invalid transaction handle.
- The transaction dealt with more than one database and a communications link fails during the rollback operation. If that happens, subtransactions on the remote node will end up in limbo. You must use **gfix** (or a functionally equivalent utility of your own design) to roll back those transactions.

  See Chapter 4 for more information about transactions that involve multiple databases.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

transaction_handle Identifier for the transaction you want to roll back. A call to **gds_$start_transaction** establishes this handle. If the call to this routine is successful, the access method sets this handle to zero. Otherwise, it leaves it unchanged.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_rollback
        if (switches & sw_commit)
          gds_$commit_transaction (status_vector, transaction);
        else
          gds_$rollback_transaction (status_vector, transaction);
```

SEE ALSO

See the entries in this chapter for:

- **gds_$commit_transaction**
- **gds_$prepare_transaction**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$send −send message

SYNOPSIS

status = **gds_$send** (
status_vector.vector_long.out,
request_handle.ulong.in,
message_type.ushort.in,
message_length.ushort.in,
message_address.unspec.in,
instantiation.ushort.in)

DESCRIPTION

The **gds_$send** routine transfers a formatted message from the calling program to the access method.

You must match each call to **gds_$send** with a **blr_receive** statement that processes the same message type. If you do not have a corresponding **blr_receive** statement in the request, the access method returns a synchronization error and unwinds the request.

Under some circumstances, your program may need to send a choice of messages (for example, ‘‘here’s some data,’’ ‘‘skip this one,’’ or ‘‘punt’’). In this case, the **gds_$send** call should be matched by a **blr_select** statement. The **blr_select** statement is a case statement containing **blr_receive** statements for different message types and actions for each type.

Remember that when coding requests each send/receive causes execution on each side to stall until the other has completed. This stalling and network traffic can be costly during remote access.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

request_handle Identifier for the request that defined the message. A call to **gds_$compile_request** establishes this handle, and a call to **gds_$start_request** or **gds_$start_and_send** activates the request.

message_type Flag that identifies type of message in message buffer. The **blr_message** statement provides the message type and structure for the message.

message_length
message_address Length in bytes and address of the calling program’s message buffer. The **blr_message** statement provides the message type and structure for the message. Your program must provide a corresponding buffer.

instantiation Incarnation of the request. A recursive routine may actually involve several active versions of the same request. This parameter specifies which instance should receive your call.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_send
        EXEC_send (message)
            MSG    message;
        {
        /************************************
         *
         *  E X E C _ s e n d
         *
         ************************************
         *
         * Functional description:
         *  Send a message to a running request.
         *  Somebody else already filled the message
         *   with good and valuable information.
         *
         ************************************/
        REQ  request;
        long  status_vector [20];

        request = message->msg_request;

        if (gds_$send (status_vector,
            request->req_handle,
            message->msg_number,
            message->msg_length,
            *message->msg_buffer),
            message->req_instance))
          db_error (request, status_vector);
        }
```

SEE ALSO

See the entry in this chapter for:

- **gds_$receive**

See also the entries in Chapter 10 for:

- **blr_receive**
- **blr_message**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$start_and_send −start request and send message

SYNOPSIS

status = **gds_$start_and_send** (
status_vector.vector_long.out,
request_handle.ulong.in,
transaction_handle.ulong.in,
message_type.ushort.in,
message_length.ushort.in,
message_address.unspec.in,
instantiation.ushort.in)

DESCRIPTION

The **gds_$start_and_send** routine is exactly equivalent to a call to **gds_$start_request** followed by a call to **gds_$send**. See the manual pages for those routines for more information about their purposes and uses.

Many requests begin with a send, transmitting variable data required for record selection. A call to **gds_$start_and_send** ‘‘piggy-backs’’ the send with the start request call. It has exactly the effect of a call to **gds_$start_request** followed by a call to **gds_$send**, thus reducing communication costs.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

request_handle Identifies the compiled request you want to start. A call to **gds_$compile_request** establishes this handle.

transaction_handle Identifies the transaction in which you want the request to execute. A call to **gds_$start_transaction** establishes this handle.

message_type Flag that identifies type of message in message buffer. The **blr_message** statement provides the message type and structure for the message.

message_length
message_address Length in bytes and address of the calling program’s message buffer. The **blr_message** statement provides the message type and structure for the message. Your program must provide a corresponding buffer.

instantiation Incarnation of the request. If you call **gds_$start_request** or **gds_$start_and_send** with the handle of an active request, that request is unwound, and a new request starts. Occasionally, in a recursive routine, you need multiple copies of the same request active simultaneously. An organization chart, for example, or a bill of materials, can most easily be done by getting a tag from one record and using it to get the next record down the tree. Instantiation numbers allow you to clone an active request and start a new version without recompiling the request.

Instantiations normally start at 0 and increase as you recurse further. The instantiation call parameter corresponds to the **level** request option in

EXAMPLE

```
. /gds/harrison/work/call_int/sands
        EXEC_start_request (request, message, level)
          REQ  request;
          MSG  message;
          SHORT  level;
        {
        /************************************
         *
         *  E X E C _ s t a r t _ r e q u e s t
         *
         ************************************
         *
         * Functional description:
         *  Start a request running.  If there is a message
         *  to send, do a start and send. Otherwise, just do
         *  a start.  If there is a message, somebody has
         *  already filled it for us.  Save the request level
         *  everywhere plausible.
         *
         ************************************/
        long status_vector [20];

        if (message)
          {
          message->msg_level = req->req_level = level;
          if (!gds_$start_and_send (status_vector,
            request->req_handle,
            request->req_database->dbb_transaction,
            message->msg_number,
            message->msg_length,
            *message->msg_buffer,
            level))
          return;
          }
        else
          {
          request->req_level = level;
          if (!gds_$start_request (status_vector,
            request->req_handle,
            request->req_database->dbb_transaction,
            level))
          return;
          }
        db_error (request, status_vector);
        }
```

SEE ALSO

See the entries in this chapter for:

- **gds_$start_request**
- **gds_$send**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$start_multiple −begin transaction

SYNOPSIS

```
status = gds_$start_multiple (
status_vector.vector_long.out,
transaction_handle.ulong.inout,
db_handle_count.ushort.in,
teb_vector_address.usbyte.in)
```

DESCRIPTION

The **gds_$start_multiple** routine begins a new transaction. It is functionally equivalent to **gds_$start_transaction**, but is intended for use with languages that have problems handling a variable number of arguments on a call. This routine can also be used in other languages when you must code the start transaction function before you know how many databases you will access.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

transaction_handle Identifier returned by this routine. The handle must be zero on input.

db_handle_count Number of database handles passed in this call. Identifies the database. This handle is returned from a call to **gds_$attach_database** or **gds_$create_database**.

teb_vector_address Address of a vector of *transaction existence blocks*, or *teb*s. The *teb_vector* is equivalent to the list of database handles and *tpb* blocks in the **gds_$start_transaction** call. The length of the *teb_vector* is determined by the *db_handle_count* parameter. There is one *teb* block in the vector for each database. The format of the *teb* block follows:

```
typdef struct {
&gds_$handle
long
*char
}  GDS_$TEB
```

dbb_ptr Address of the database handle.

tpb_len Length of the transaction parameter block for that database.

tpb_ptr Pointer to a transaction parameter block (*tpb*).

tpb_vector Describes the conditions of access. The *tpb_vector* consists of a version number and a vector of bytes that describes the transaction characteristics:

tpb_vector ::= *version_number*.ubyte vector_byte
version_number ::= **tpb_$version3**

See the section below titled ‘‘TPB PARAMETERS’’ for a list of values for *vector_byte*.

TPB PARAMETERS

The following are valid parameter values:

gds_$tpb_concurrency (default)
gds_$tpb_consistency The default mode for a transaction specifies a high throughput, high concurrency transaction with generally acceptable consistency. The optional mode specifies that the operations performed in the transaction should be serializable in some order.

gds_$tpb_wait (default)
gds_$tpb_nowait The default action if your program encounters a locked relation is to wait until the lock goes away. The nowait option is not recommended.

gds_$tpb_write (default)
gds_$tpb_read The default intention of a transaction is that it will write data. Both the default and the option take a relation name as an argument.

gds_$tpb_lock_level Specifies the intention of a transaction toward a specified relation. The format of *gds_$tpb_lock_level* is:

lock_option, length, relation_name, access_option

lock_option ::= { **gds_$tpb_lock_read** | **gds_$tpb_lock_write** }

access_option ::= { **gds_$tpb_shared** | **gds_$tpb_protected** | **gds_$tpb_exclusive** }

The default *lock_option* is write. The default *access_option* is concurrent shared access, the protected option allows concurrent restricted access, and the exclusive option disallows any concurrent access.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_teb
        #include "/sys/ins/gds.ins.c"

        typedef struct {
          int  *dbb_ptr;
          long  tpb_len;
          char  *tpb_ptr;
        }  GDS_$TEB;

        GDS_$TEB  teb_vec [2];
```

```
int
  gds_$status[20], /* status vector */
  *db0, *db1, /* database handle */
  *trans;

static char
  gds_$tpb_0 [] = {
    gds_$tpb_version3, gds_$tpb_write,
    gds_$tpb_consistency, gds_$tpb_wait,
    gds_$tpb_lock_write, 3,'I','D','S',
    gds_$tpb_protected},

  gds_$tpb_1 [] = {
    gds_$tpb_version3, gds_$tpb_write,
    gds_$tpb_consistency, gds_$tpb_wait,
    gds_$tpb_lock_write, 3,'O','Z','S',
    gds_$tpb_protected};

main ()
{
db0 = db1 = 0;
trans = 0;

if (! gds_$attach_database (
    gds_$status, 0, "test_0.gdb", db0, 0,0))
  gds_$attach_database (
    gds_$status, 0, "test_1.gdb", db1, 0,0);

if (db0 && db1)
  {
  teb_vec[0].dbb_ptr = &db0;
  teb_vec[0].tpb_len = sizeof (gds_$tpb_0);
  teb_vec[0].tpb_ptr = gds_$tpb_0;

  teb_vec[1].dbb_ptr = &db1;
  teb_vec[1].tpb_len = sizeof (gds_$tpb_1);
  teb_vec[1].tpb_ptr = gds_$tpb_1;

  if (gds_$start_multiple (gds_$status, trans, 2, teb_vec))
  gds_$print_status (gds_$status);
  }

if (trans)
  gds_$commit_transaction (gds_$status, trans);

if (db0 && !trans)
  gds_$detach_database (gds_$status, db0);
```

```
if (db1 && !(trans && db0))
  gds_$detach_database (gds_$status, db1);

if (gds_$status [1])
  gds_$print_status (gds_$status);

}
```

SEE ALSO

See the entries in this chapter for:

- **gds_$start_transaction**
- **gds_$prepare_transaction**
- **gds_$commit_transaction**
- **gds_$rollback_transaction**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$start_request −begin request

SYNOPSIS

```
status = gds_$start_request (
status_vector.vector_long.out,
request_handle.ulong.in,
transaction_handle.ulong.in,
instantiation.ushort.in)
```

DESCRIPTION

The **gds_$start_request** routine begins the execution of a previously compiled request under an existing transaction.

You can attach a database, start a transaction, compile a request, or get information about anything, but until you start a request, you cannot touch the contents of a database. Compiling a request prepares an executable form of the request. A call to **gds_$start_request** begins execution.

If you call **gds_$start_request** and pass the handle of a request that is already running, the access method unwinds the active request and starts a new request. However, if you pass a value for *instantiation*, the access method starts a new instance of that request.

At times, particularly in recursive code, you may want to run several copies of the same request. You cannot simply restart the request because it has internal context that it must retain. You can, however, start another instance of the request, using the *instantiation* parameter on the **gds_$start_request** call. A start request call with an unused value for the *instantiation* parameter clones the request, giving it the same executable structure but with a separate context. Creating a new instance of a request is considerably less expensive than recompiling the BLR to create a new request.

Many requests begin with a send, transmitting variable data required for record selection. A call to **gds_$start_and_send** ‘‘piggy-backs’’ the send with the start request call. It has exactly the effect of a call to **gds_$start_request** followed by a call to **gds_$send**, and reducing communication costs.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

request_handle Identifies the compiled request you want to start. A call to **gds_$compile_request** establishes this handle.

transaction_handle Identifies the transaction in which you want the request to execute. A call to **gds_$start_transaction** establishes this handle.

instantiation Incarnation of the request. If you call **gds_$start_request** or **gds_$start_and_send** with the handle of an active request, that request is unwound, and a new request starts. Occasionally, in a recursive routine, you need multiple copies of the same request active simultaneously. An organization chart, for example, or a bill of materials, can most easily be done by getting a tag from one record and using it to get the next record down the tree. Instantiation numbers allow you to clone an active request and start a new

version without recompiling BLR.

Instantiations normally start at 0 and increase as you recurse further. The instantiation call parameter corresponds to the **level** request option in GDML.

EXAMPLE

The following example actually takes advantage of piggybacking of a start request and send through the **gds_$start_and_send** routine:

```
. source: /gds/harrison/work/call_int/sands
        EXEC_start_request (request, message, level)
            REQ    request;
            MSG    message;
            SHORT       level;
        {
        /************************************
         *
         *  E X E C _ s t a r t _ r e q u e s t
         *
         ************************************
         *
         * Functional description:
         *  Start a request running.  If there is a message
         *  to send, do a start and send, otherwise just do
         *  a start.  If there is a message, somebody has
         *  already filled it for us.  Save the request level
         *  everywhere plausible.
         *
         ************************************/
        long  status_vector [20];

        if (message)
          {
          message->msg_level = req->req_level = level;
          if (!gds_$start_and_send (status_vector,
            request->req_handle,
            request->req_database->dbb_transaction,
            message->msg_number,
            message->msg_length,
            *message->msg_buffer,
            level))
          return;
          }
        else
          {
          request->req_level = level;
          if (!gds_$start_request (status_vector,
            request->req_handle,
            request->req_database->dbb_transaction,
```

```
    level))
  return;
  }
db_error (request, status_vector);
}
```

SEE ALSO

See the entries in this chapter for:

- **gds_$compile_request**
- **gds_$start_and_send**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$start_transaction −begin transaction

SYNTAX

```
status = gds_$start_transaction (
status_vector.vector_status.out,
transaction_handle.ulong.inout,
db_handle_count.ushort.in,
{ db_handle.ulong.in,
tpb_length.ushort.in,
tpb_address.ubyte.in }... )
```

DESCRIPTION

The **gds_$start_transaction** routine begins a new transaction. See the Chapter 4 of this manual for transaction capabilities accessible only through the call interface.

If you are working with multiple databases and a language that requires a fixed number of arguments on each call, you should call **gds_$start_multiple** instead of **gds_$start_transaction**. The **gds_$start_multiple** routine is also useful if you need to code the start transaction call without knowing how many databases will be involved.

For most applications, you may find that the **start_transaction** statement is easier to code than a call to **gds_$start_transaction**. The statement and the start transaction call are functionally equivalent.

If the module does not contain any other statements, calling the **gds_$start_transaction** routine will avoid the extra step of preprocessing the program with **gpre**.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

transaction_handle Identifier returned by this routine. The handle must be zero on input.

db_handle_count Number of database handles passed in this call. Because a single transaction can access multiple databases, this routine passes information about each database it accesses and the conditions of access for that database:

db_handle Identifies the database.

tpb_length Length of the *tpb_vector* that describes the conditions of access.

tpb_address Address of the *tpb_vector*.

The *tpb_vector* (*t*ransaction *p*arameter *b*lock) consists of a version number and a vector of bytes that describes the transaction characteristics:

tpb_vector ::= *version_number*.ubyte vector_byte

version_number ::= **tpb_$version3**

See ‘‘TPB Parameters below for a list of values for *vector_byte*.

TPB PARAMETERS

The following are valid parameter values:

gds_$tpb_concurrency (default)
gds_$tpb_consistency The default mode for a transaction specifies a high throughput, high concurrency transaction with generally acceptable consistency. The optional mode specifies that the operations performed in the transaction should be serializable in some order.

gds_$tpb_wait (default)
gds_$tpb_nowait The default action if your program encounters a locked relation is to wait until the lock goes away. The nowait option is not recommended.

gds_$tpb_write (default)
gds_$tpb_read The default intention of a transaction is that it will write data. Both the default and the option take a relation name as an argument.

gds_$tpb_lock_level Specifies the intention of a transaction toward a specified relation. The format of *gds_$tpb_lock_level* is:

lock_option, length, relation_name, access_option

lock_option ::= { **gds_$tpb_lock_read** | **gds_$tpb_lock_write** }

access_option ::=
{ **gds_$tpb_shared** | **gds_$tpb_protected** | **gds_$tpb_exclusive** }

The default *lock_option* is write. The default *access_option* is concurrent shared access, the protected option allows concurrent restricted access, and the exclusive option disallows any concurrent access.

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_23
        #include "/sys/ins/gds.ins.c"

        int
          gds_$status[20], /* status vector */
          *db, /* database handle */
          *trans;

        static char
          gds_$tpb_0 [] = {
            gds_$tpb_version3, gds_$tpb_write,
            gds_$tpb_consistency, gds_$tpb_wait,
            gds_$tpb_lock_write, 3,'I','D','S',
```

```
    gds_$tpb_protected};

main ()
{
db = 0;
trans = 0;

gds_$attach_database (gds_$status, 0, "test.gdb", db, 0,0);

if (db)
  if (gds_$start_transaction (
    gds_$status, trans, 1, db,
    sizeof(gds_$tpb_0), gds_$tpb_0))
  gds_$print_status (gds_$status);
if (trans)
  gds_$commit_transaction (gds_$status, trans);

if (db && !trans)
  gds_$detach_database (gds_$status, db);

if (gds_$status [1])
  gds_$print_status (gds_$status);

}
```

SEE ALSO

See the entries in this chapter for:

- **gds_$prepare_transaction**
- **gds_$commit_transaction**
- **gds_$rollback_transaction**
- **gds_$start_multiple**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$transaction_info −information call

SYNOPSIS

status = **gds_$transaction_info** (
status_vector.vector_long.out,
transaction_handle.ulong.in,
item_list_length.ushort.in,
item_list_buffer_address.vector_byte.in,
result_buffer_length.ushort.in,
result_buffer_address.unspec.out)

DESCRIPTION

The **gds_$transaction_info** routine returns information about the current transaction. A call to **gds_$transaction_info** returns information necessary for keeping track of permanent transaction ids, instead of volatile transaction handles.

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program. If you pass zero as the address of the status vector and encounter an error, writes the message(s) to standard error and aborts your program. See Chapter 8 for a discussion of the status vector.

The returned status code indicates only that the access method accepted the request for information; it does not mean that it understood the request or that it supplied all requested information. Your program must interpret the contents of the result buffer.

transaction_handle Identifies the transaction about which you would like some information.

item_list_buffer_length
item_list_buffer_address Provides the length and address of the item list buffer. The item list buffer is a regular byte vector with no structure. The calling program lists the items about which it requires information in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

result_buffer_length
result_buffer_address Provides the length and address of the result buffer. The access method returns the requested information to the result buffer. The result buffer has the following format:

Syntax: Information Call Result Buffer Clumplet

result_buffer ::= clumplet...

clumplet ::= *type*.ubyte *length*.ushort *value*.short

The value of *type* is the item you requested in the item list buffer. These items are listed below under the heading ‘‘Information Items.’’

The clumplets returned to the result buffer are not aligned. Furthermore, binary numbers are in a generic format, which you must convert to a datatype native to your computer before interpreting them. In a generic binary value, the least significant byte is first, and the most significant is last. The sign is in the last byte. To interpret a binary value returned by an information call:

- Determine the size, which can be *1*, *2*, or *4* bytes.
- Reverse the order of the bytes.

The following routine converts the contents of the result buffer into something you can read:

```
REV_integer (ptr, length)
          unsigned char *ptr;
          short length;
        {
        /**************************************
         *
         *  R E V _ i n t e g e r
         *
         **************************************
         *
         * Functional description:
         *  Pick up (and convert) an integer
         *  of length 1, 2, or 4 bytes.
         *
         **************************************/
        int     value;
        short   shift;

        value = shift = 0;

        while (--length >= 0)
          {
          value += (*ptr++) << shift;
          shift += 8;
          }

        return value;
        }
```

INFORMATION ITEMS

You can ask about the following item in the item list buffer:

gds_$info_tra_id Transaction id number. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_tra_id**
length	ushort	length of clumplet in bytes
value	short	transaction id

In addition to the above item about which you can request information, the access method may also return the following status messages to the result buffer:

gds_$info_end End of message. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_end**

gds_$info_truncated Input into the result buffer was truncated. The access method returns a truncated clumplet as the last clumplet in the result buffer if the result buffer was not large enough to hold all the information you requested. If your program encounters this clumplet, it means that all preceding information is valid, but at least one item is missing. The packet returned to the result buffer has the following format:

type	ubyte	**gds_$info_truncated**

gds_$info_error An error. The access method returns an error clumplet if an item of requested information was not available. This clumplet has the same form as other clumplets, but the information portion contains only the information type value and a code indicating why the information was not available. The packet returned to the result buffer has the following format:

type	ubyte	*gds_$info_error*

EXAMPLE

```
. source: /gds/harrison/work/call_int/examp_tra_info
        static char  tra_items [] =
          {gds_$info_tra_id}

        CHAR tra_info [32];


        if (gds_$transaction_info (status_vector,
            blob,
            sizeof (tra_items),
            tra_items,
            sizeof (tra_info),
            tra_info))
```

```
error ("gds_$transaction_info failed", status_vector);
```

SEE ALSO

See the entry in this chapter for:

- **gds_$reconnect_transaction**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

gds_$unwind_request −stop running request

SYNOPSIS

status = **gds_$unwind_request** (
status_vector.vector_long.out,
request_handle.ulong.in,
instantiation.ushort.in)

DESCRIPTION

The **gds_$unwind_request** routine stops a running request. The target request stops as soon as it can stop safely. When you unwind an instantiated request, the access method also unwinds all instantiations of that request with higher values for *instantiation*.

A call to this routine lets you terminate a request cleanly without aborting the transaction or database attach. It is particularly useful in interactive applications where the user provides selection criteria that return more records than desired.

Any updates made prior to this call will remain in the database unless you roll back the transaction with a call to the **gds_$rollback_transaction** routine.

Although a request has been unwound, it can be re-used without being recompiled. You can re-use the transaction by passing the request handle of the request you want to start to the **gds_$start_request** routine.

In addition to a successful return from a call to this routine, the following events cause a request to unwind:

- An error
- A **blr_leave** statement
- Restart of the request by **gds_$start_request**
- Release of the request by **gds_$release_request**
- Powerdown, hardware fault, operating system crash, or other system failure
- A successful return from **gds_$commit_transaction**, **gds_$rollback_transactionf**, and **gds_$detach_database**

PARAMETERS

status_vector A vector of 20 longwords that the access method uses to return error messages to the calling program.

If you pass zero as the address of the status vector and encounter an error, the access method writes the message(s) to standard error and aborts your program.

request_handle Identifies the request you want to unwind. A call to **gds_$unwind_request** establishes this handle, and a call to **gds_$start_request** or **gds_$start_and_send** activates the request.

instantiation Incarnation of the request. The instantiation parameter in this call identifies the instance of request you want to unwind.

EXAMPLE

```
EXEC_stop ()
        {
        /************************************
         *
         *  E X E C _ s t o p
         *
         ************************************
         *
         * Functional description:
         *  A loop has been stopped.  Unwind active
         *  requests set flag.
         *
         ************************************/
        long status_vector [20];
        REQ request;

        for (request = TOP_request; request; request = request->req_next)
          if (request->req_handle)
            gds_$unwind_request (status_vector,
              request->req_handle,
              request->req_level);

        SIG_stop = TRUE;
        }
```

SEE ALSO

See the entries in this chapter for:

- **gds_$rollback_transaction**
- **gds_$start_request**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.