Part II: Generic Interface

3. Introduction

This part of the GDI User Manual describes the functions that provide the following capabilities:

- Database communications
- SQL query execution
- Transaction Management
- Error handling
- Specialized database functions, such as unique key assignment and data dictionary access

3.1 Stylistic and Naming Conventions

The following general principles and naming conventions are followed throughout the GDI:

Libraries: Currently the GDI consists of several libraries. The main module, *libgdi.a*, must be linked with the desired vendor-specific library (or libraries, if more than one database is accessed). Vendor-specific libraries are *libgdiora.a* for ORACLE, *libgdipg.a* for POSTGRES, and *libgdisyb.a* for SYBASE.

Include Files: C applications must include *libgendb.h*. FORTRAN applications must include *libgdifor.h*.

Functions: All generic database functions are named “gdi_<function_name>”.

Macros: All macros are in uppercase and named “GDI_<macro_name>”.

Typedefs: All generic database type definitions are named “db<TypedefName>”.

Compilation: *libgdi.a* is compiled with the regular Unix C compiler, **cc**, for installation but has been compiled previously with **gcc** for error checking.

Prototypes: All functions are prototyped in header files. Prototypes include argument types and argument names if __STDC__ is defined.

Man Pages: Man pages are listed with the man page section to which they belong. For example, gdi_gen_Astructs(1) is a Section 1 man-page. All *libgdi.a* functions are in Section 3.

3.2 Location of GDI Components

Table 1. summarizes the location of GDI components. *INSTALL_DIR* refers to the directory tree where software is normally installed for production access. *LIBSRC* refers to the directory containing library source code.

Table 1. Summary of Locations

Name	*Description*	*Directory Location*
libgdi.a	Library that GDI applications link.	*INSTALL_DIR*/lib
libgendb.h, libgdifor.h	Public GDI header that applications include in source code files.	*INSTALL_DIR*/include
GDI man pages	Describes each gdi_* function call.	*INSTALL_DIR*/man
gdi_gen_Astructs	Header generator for ArrayStructs tuple constructor; see gdi_gen_Astructs(1).	*INSTALL_DIR*/bin
source code	GDI functions.	*LIBSRC*/libgendb/src
unit tests and sample code	Unit tests that exercise and demonstrate GDI functions.	*LIBSRC*/libgendb/test
FORTRAN unit tests	Unit tests that exercise and demonstrate the FORTRAN interface.	*LIBSRC*/libgendb/test
GDI User Manual	FrameMaker source organized into a book named *gdi.bk*.	*LIBSRC*/libgendb/doc/fm/user_manual

3.3 Sample Programs

The programs in *LIBSRC*/libgendb/test exercise *libgdi.a* functions and constitute sample code that demonstrates, how to use GDI functions. Table 2 summarizes the test programs.

Table 2. GDI Sample Programs

Program	*Description*
interact_submit	Tests the *gdi_submit()* function by prompting for input interactively.
tst_ArrayStructs_submit tst_ArrayStructs_insert	Tests the ArrayStructs tuple constructor, which manages data in an array of structures.
tst_conn	Tests database connect functions.
tst_constr	Tests constructor functions.
tst_create	Creates a temporary table in the database.
tst_dbobj	Tests dbObj functions.
tst_get_counter	Tests the *gdi_get_counter()* routine.
tst_get_dbcount	Tests Oracle PRO*C hooks, requires database open with *oracle_open()*.
tst_insert1	Fetches data from the database and inserts it into another table in the database.
tst_insert2	Creates a dbObj and populates it with data that it then inserts into the database.
tst_submit	Tests the *gdi_submit()* function.
tst_whatis	Tests the *gdi_what_is_object()* function.

Each program uses the Caltech *libpar.a* library to parse command line arguments and has a par file in the *LIBSRC*/libgendb/test directory. Additional par files are in *LIBSRC*/libgendb/test/par. These par files access project-specific databases used during GDI development and testing. They should be checked to make sure database connect information and operations are appropriate for the local database.

Instructions for compiling and executing each test stub are based on the source code filename (Table 3).

Table 3. Test Stub Instructions

	General Instructions	*Example*
Source Code	*program_name*.c	tst_conn.c
Par File	*program_name*.par	tst_conn.par
To Compile	make *program_name*	make tst_conn
To Execute	*program_name* par=*program_name*.par	tst_conn.par=tst_conn.par

3.4 Database-Specific Notes

3.4.1 ORACLE

3.4.1.1 Compiling Applications

Applications must link *libgdi.a* with an ORACLE -specific library, *libgdiora.a*, and with ORACLE libraries at revision 6.0.36.4 or higher because new OCI functions used by the GDI became available in that release. As of this writing, the following 6.0.36.4 libraries must be linked (see the sample Makefile in *LIBSRC*/libgendb/test):

liboci14c.a	OCI routines
libsql14.a	PRO*C routines
libsqlnet.a	SQL*Net library
libora.a	ORACLE RDBMS kernel routines

Once compiled with 6.0.36.4, the application may be used with ORACLE databases running an earlier revision. It has been used extensively with 6.0.33.2 databases.

*3.4.1.2 Support for PRO*C Routines*

Currently*, gdi_open()* establishes a database connection with OCI. This allows multiple, concurrent connections. Applications may link in their own PRO*C subroutines; but they must first establish a PRO*C database connection with the GDI function *oracle_open* (see oracle_-open(3)). PRO*C subroutines must be executed on that connection. Due to a limitation of Oracle version 6, only one PRO*C connection may be opened. However, additional OCI connections may be established with *gdi_open*. A future enhancement will allow multiple PRO*C opens working properly.

A low level error handling routine, *ora_sqlca_error()*, provides developers of PRO*C routines with the ability to store SQLCA error information in the dbObj (see ora_sqlca_error(3)). Sample calling syntax is:

```
EXEC SQL OPEN my_cursor;
if (ora_sqlca_error (conn, sqlca, “my_cursor open: “) != GDI_SUCCESS)
        return (GDI_FAILURE);
```

ora_count.pc in *LIBSRC*/libgendb/test tests the PRO*C hooks. The *tst_get_dbcount.c* file in *LIBSRC*/libgendb/test exercises the PRO*C function.

3.4.1.3 Calculated Numbers are Doubles

Calculated columns will be returned as doubles, even if the result is an integer. For example, the following query will return *count* as a double:

```
select count(wfid) count from wfdisc where wfid > 50000
```

3.4.1.4 Fixed Date Format

The default ORACLE date format contains only the date (year, month, day); it does not include time (hours, minutes, seconds). Version 6 does not allow setting a different default date format; although, that capability will be available in ORACLE Version 7. Until Version 7 becomes widely available, the following ORACLE date format will be expected throughout the GDI:

```
YYYYMMDD HH24:MI:SS
```

Later versions of the GDI will be able to support user-defined date formats.

3.4.1.5 Link System V

Developers can compile applications any way they like, but the final link must be System V rather than BSD. If a segmentation fault occurs on a database select inside a lower level ORACLE routine, the application is probably resolving symbols from */usr/lib/libc.a* instead of */usr/5lib/libc.a*.

3.4.2 POSTGRES

Work in progress.

3.4.3 SYBASE

Work in progress.

4. Query Management (dbConn)

Table 4. Summary of Query Channel Functions

Name	*Description*	*Man Page*	*Sample Code*
gdi_init	Initialize the GDI library	none yet	tst_conn.c
gdi_open	Establishes a connection to the database.	gdi_open(3)	tst_conn.c
gdi_close	Closes a connection to the database.	gdi_close(3)	tst_conn.c
gdi_exit	Closes all database connections.	gdi_exit(3)	tst_conn.c
gdi_dead	Checks to see if connection is live.	gdi_dead(3)	
gdi_print_conn	Outputs contents of dbConn to *stdout*.	gdi_print_conn(3)	tst_conn.c
oracle_open	Opens and Oracle PRO*C connection	oracle_open(3)	tst_get_dbcount.c

4.1 Connecting to a Database

gdi_init() initializes the GDI library. It takes the executable filename as a parameter*. gdi_init()* should be called once by the application program before any other GDI functions are called.

Example 1:

```
gdi_init (argv[0]);
```

gdi_open() connects a process to a database and returns a dbConn structure. A NULL dbConn means the connect failed. Example 2 shows how a program called SampleProgram might connect to an ORACLE database.

Example 2:

```
dbConn      *my_dbConn1;
char        *vendor=”oracle”;
char        *account=”scott”;
char        *password=”tiger”;
char        *db=”t:troll:dev”;              /* ORACLE Version 6 SQL*Net TWO_TASK string */

if ((my_dbConn1 = gdi_open (vendor, account, password, db, NULL, NULL) == (dbConn *) NULL)
{
      ... handle error ...
}
```

The last two *gdi_open()* parameters are NULL because they are not used for connecting to ORACLE. Also, if the *account* parameter contains the entire ORACLE connect string, the rest of the parameters may be left NULL. Example 3 would create the same database login as Example 2.

Example 3:

```
dbConn      *my_dbConn1;
char        *vendor=”oracle”;
char        *account=”scott/tiger@t:troll:dev”;

if ((my_dbConn1 = gdi_open (vendor, account, NULL, NULL, NULL, NULL) == (dbConn *) NULL)
{
      ... handle error ...
```

```
}
```

At this point, SampleProgram is now connected to one database, as depicted in Figure 6.

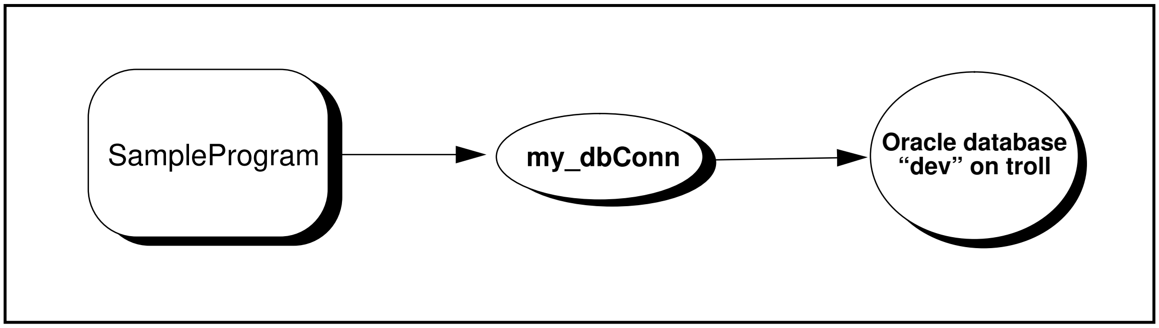

FIGURE 6. SampleProgram Connected to one Database

An application may connect to more than one database simultaneously. Example 4 shows the same process connecting to a POSTGRES database.

Example 4:

```
dbConn     *my_dbConn2;
char       *vendor=”postgres”;
char       *account=NULL;
char       *password=NULL
char       *db=”geodemo”;
char       *server=NULL;
char       *app=NULL;

if ((my_dbConn2 = gdi_open (vendor, account, password, db, server, app) == (dbConn *) NULL)
{
        ... handle error ...
}
```

The database host will be driven by the POSTGRES PGHOST environmental variable. SampleProgram is now connected to two databases, as depicted in Figure 7.

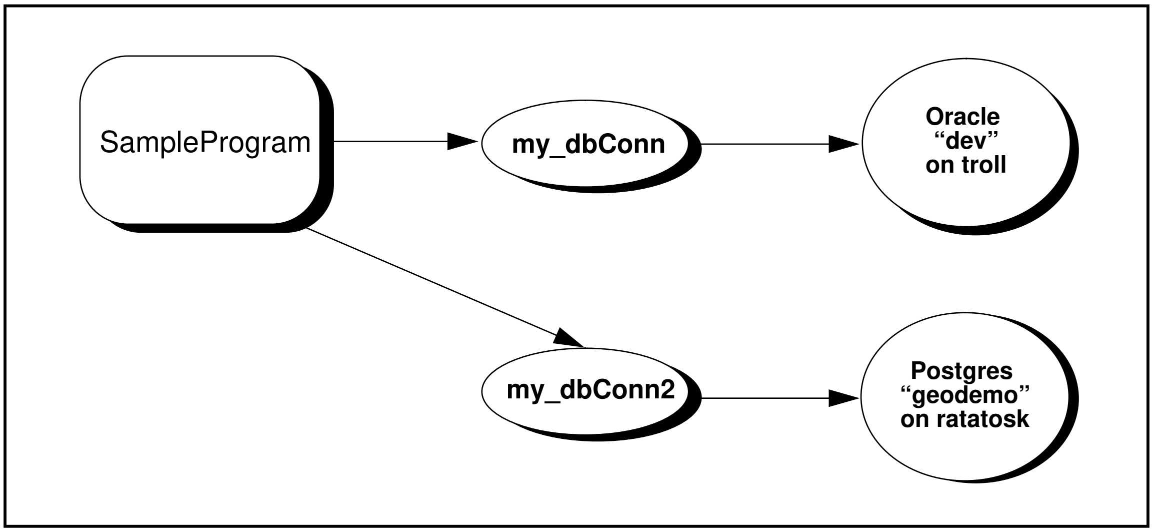

FIGURE 7. SampleProgram Connected to Two Databases

Each dbConn keeps track of database login information, error information and some vendor-specific information. The contents of the dbConn may be output with *gdi_print_conn().* Example 5 shows how the dbConn connections established by Example 3 and Example 4 could be output to *stdout*.

Example 5:

```
gdi_print_conn (my_dbConn);
gdi_print_conn (my_dbConn2);
```

The connection to the database could be broken for a variety of reasons (network down or too unreliable to sustain a connect, database down, database host crashed, just to name a few). *gdi_dead()* determines if a dbConn is still alive (currently only implemented in POSTGRES).

Example 6:

```
if (gdi_dead (my_dbConn) == TRUE)
{
        ... connection dropped, do something appropriate ...
}
```

gdi_close() closes a specific database connection. Example 7 closes *my_dbConn*; but *my_dbConn2* remains open.

Example 7:

```
gdi_close (my_dbConn);
```

gdi_exit() closes all open connections. Example 8 closes both *my_dbConn* and *my_dbConn2*.

Example 8:

```
gdi_exit ();
```

Table 4. summarizes dbConn functions

4.2 Managing Query Channels

In addition to storing login and error information, the dbConn also tracks query channels, “pipes” on which database commands get executed.

Query channels are analogous to UNIX shells:

- *UNIX shell*
  After logging into a UNIX workstation, a user executes UNIX commands in a shell. The workstation might be running a window manager such as Motif that allows creating additional windows. Used together, multiple windows make the job at hand more efficient. The user might read email in one window that asks which *ps* option lists the virtual size of processes. The user can run *man ps* in a second window to verify that the option is *v*, then actually run *ps v* in a third window so the output can be cut and pasted in an email response in the first window. The UNIX login to the workstation keeps track of the shells. If the login goes away, all the shells disappear.
- *Database query channel*
  After logging into a database, a process executes database commands on a query channel. GDI functions allow creating of additional channels. One channel might be used to read a large amount of data from the database. A second channel might update a table based on information read from the first. The dbConn keeps track of the query channels. If the dbConn disappears, all the query channels disappear.

gdi_open() creates default query channels that are managed by GDI routines. If an application uses just GDI routines, it does not need to do anything with query channels.

Applications that add database routines may need to know about query channels, information provided by the rest of this section.

Each channel equates to a cursor for ORACLE, to a portal for POSTGRES (if a fetch is involved), and to a DBPROCESS for SYBASE. *gdi_open()* creates two query channels with the loose notion that one is for reading, the other for writing. *libgendb.h* defines aliases for accessing these two channels. The first channel may be used by specifying GDI_DEFAULT_CHAN or GDI_SELECT_CHAN. The second may be used by specifying GDI_UPDATE_CHAN.

The GDI attempts to provide consistent handling across databases, but this is not always possible. Sometimes a query channel makes sense for one database but not another. For example, ORACLE manages transactions at the dbConn level while SYBASE manages them at the channel level. Example 8 shows how variable handling may be accommodated in an application.

Example 9:

```
#ifdef SYBASE
      channo = GDI_DEFAULT_CHAN;
#else
      channo = GDI_NOT_USED;
#endif
```

If a valid query channel is specified for ORACLE in a function that does not use one, it will be ignored; however, the resulting code may cause confusion with something as fundamental as transaction management.

Example 8 creates an additional query channel. Note that the *address* of the new query channel should be passed to *gdi_open_channel()*. The GDI manages a list of channels. The channel will be created and a number assigned for accessing it.

Example 10:

```
int     my_channel;

if (gdi_open_channel (my_dbConn, &my_channel) != GDI_SUCCESS)
{
        ... handle error ...
}
```

Example 8 checks to see if the channel is still open.

Example 11:

```
if (gdi_channel_is_open (my_dbConn, my_channel) != TRUE)
{
        ... handle error ...
}
```

Example 8 shows how *gdi_flush()* discards any unprocessed query results. For ORACLE, this cancels a query after the desired number of rows have been fetched and frees any resources associated with the cursor. For SYBASE, it cancels any rows pending in the DBPROCESS results buffer if the user did not process all rows in the results set. For POSTGRES, this clears the portal, if appropriate.

Example 12:

```
if (gdi_flush (my_dbConn, my_channel) != GDI_SUCCESS)
{
        ... handle error ...
}
```

gdi_abort() terminates the currently executing command. For ORACLE, if no command is currently executing and the next command is a fetch, the fetch will be aborted. For SYBASE, all commands in the current command batch are cancelled. This command has no effect for POSTGRES.

Example 8 closes the query channel created in Example 8.

Example 13:

```
if (gdi_close_channel (my_dbConn, my_channel) != GDI_SUCCESS)
{
        ... handle error ...
}
```

Table 5. summarizes query channel functions.

Table 5. Summary of Query Channel Functions

Name	*Description*	*Man Page*	*Sample Code*
gdi_open_channel	Opens an additional query channel.	gdi_open_channel(3)	tst_conn.c
gdi_close_channel	Closes the specified query channel.	gdi_close_channel(3)	tst_conn.c
gdi_channel_is_open	Checks to see if channel is still open.	gdi_channel_is_open(3)	
gdi_abort	Terminates the current command.	gdi_abort(3)	
gdi_flush	Discards unprocessed query results.	gdi_flush(3)	

4.3 Submitting Queries

gdi_submit() executes any database query. The basic sequence is:

1. Connect to the database with *gdi_open()*. Queries will be submitted on the dbConn that is returned.
2. Populate a null-terminated string with an database query. For users accustomed to ORACLE, the query should not have a terminating semi-colon (;).
3. Execute the query with *gdi_submit()*.
4. Handle any return results. If the database query is a SELECT (ORACLE and SYBASE) or RETRIEVE (POSTGRES), a dbObj will contain the results. The dbObj is described in Section 5.
5. Free the return results structure.

The test routine *tst_submit.c* has a complete example.

4.4 Specialized Database Functions

Table 6. summarizes a handful of specialized database functions.

Table 6. Summary of Specialized Database Functions

Name	*Description*	*Man Page*	*Sample Code*
gdi_get_counter	Get a unique key id.	gdi_get_counter(3)	tst_get_counter.c
gdi_what_is_object	Returns what an object is and who owns it.	none yet	tst_whatis.c
gdi_create_table	Creates a database table based on its dbObj definition.	none yet	tst_create.c

5. Data Management (dbObj)

The Database Object (dbObj) manages data and is created whenever a database query is executed. An application can also create a dbObj and store data in it, then use it to create and populate a table in the database. Its structure is defined in the *libgendb.h* include file and depicted in Figure 8.

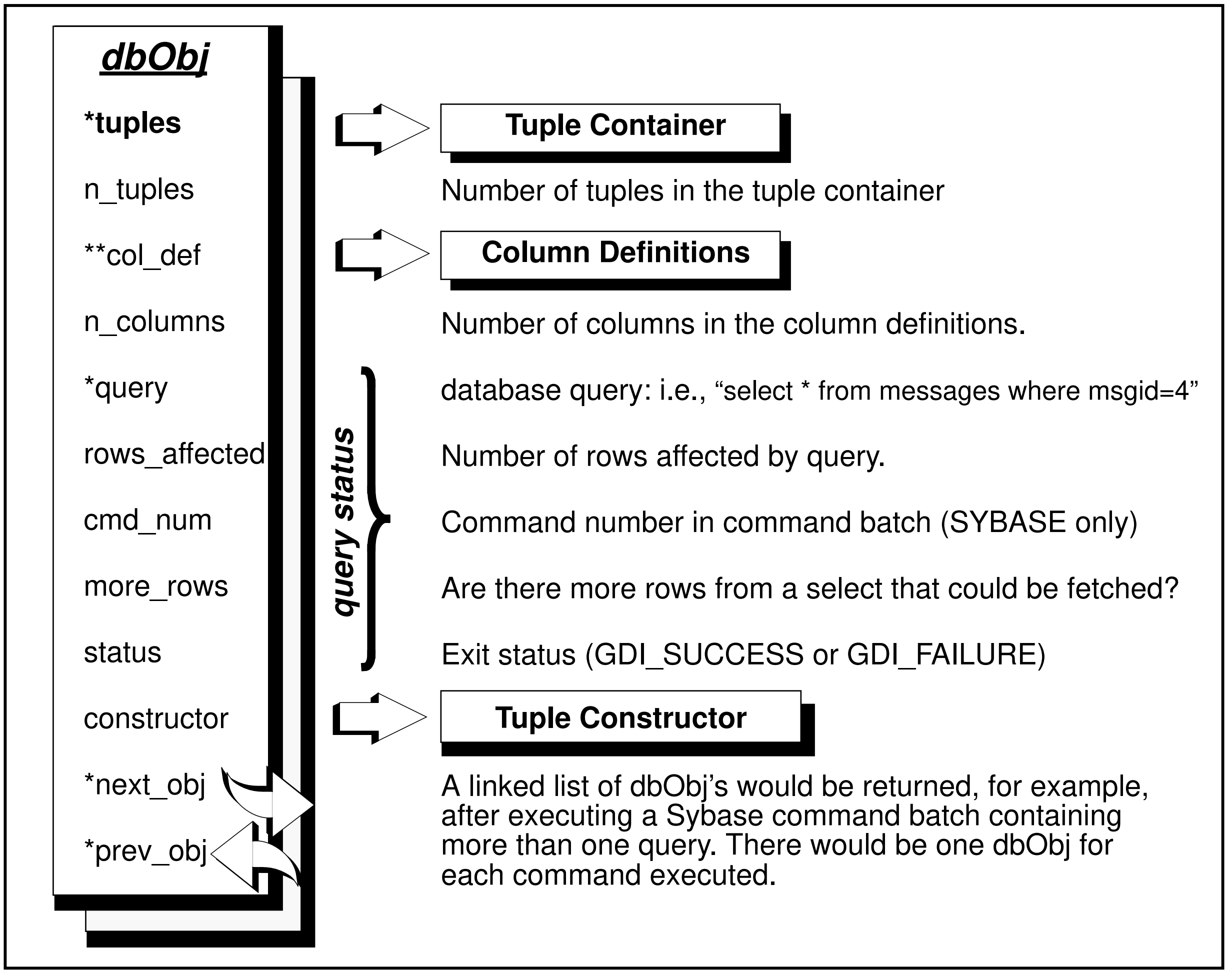

FIGURE 8. dbObj Structure

The dbObj consists of 4 basic parts:

- *Tuple Container*
  Stores query results if the query is a SELECT (ORACLE and SYBASE) or RETRIEVE (POSTGRES), or data to be inserted into the database if the query is an INSERT (ORACLE and SYBASE) or APPEND (POSTGRES).
- *Column Definitions*
  Describes each field in the rows stored in the tuple container, such as column name, data type and size.

- *Query Information*
  Several variables store miscellaneous information such as the text of the database query, the number of rows affected, and whether the function succeeded or failed.
- *Tuple Constructor*
  Controls the structure or format of the data in the tuple container.

A dbObj should never be accessed directly because the specific structure will likely change. Instead, the macros and functions summarized inTable 7 should be used. The sample code referenced in the table is in *LIBSRC*/libgendb/test.

Table 7. Summary of dbObj Macros and Functions

Name	*Description*	*Sample Code*
dbObj Creation		
gdi_obj_create	Creates a new dbObj and copies the constructor into the dbObj constructor.	tst_create.c, tst_dbobj.c, tst_insert2.c
gdi_obj_destroy	Frees a dbObj, deallocating all allocated fields.	interact_submit.c, tst_constr.c, tst_create.c, tst_dbobj.c, tst_insert1.c, tst_insert2.c, tst_submit.c, tst_whatis.c
Tuple Container		
GDI_OBJ_TUPLES	Pointer to the tuple container	
GDI_OBJ_NUM_TUPLES	Number of tuples in the tuple container.	interact_submit.c, tst_constr.c, tst_dbobj.c, tst_insert2.c, tst_-submit.c
Column Definitions		
GDI_OBJ_COL_DEFS	Pointer to an array of column definitions.	
GDI_OBJ_NUM_COLUMNS	Number of columns.	
Query Status		
GDI_OBJ_QUERY	Database query.	tst_insert1.c
GDI_OBJ_ROWS_AFFECTED	Number of rows affected by the database command.	tst_dbobj.c, tst_insert1.c, tst_insert2.c, tst_submit.c
GDI_OBJ_CMD_NUM	Command number (may be >1 for SYBASE)	
GDI_OBJ_MORE_ROWS	Indicates there were more rows to be had; *i.e.,* the number of records requested was less than the actual query results.	
GDI_OBJ_STATUS	Command status	

Table 7. Summary of dbObj Macros and Functions

Name	*Description*	*Sample Code*
Tuple Constructor		
GDI_OBJ_CONSTRUCTOR	Pointer to the tuple constructor	

5.1 Tuple Container

Programs do not need to know the actual structure of the tuples or of the tuple container. The functions summarized in Table 8 provide data access regardless of the actual structure.

Table 8. Summary of Tuple Container Macros and Functions

Name	*Description*	*Sample Code*
gdi_obj_container_create	Creates a tuple container in the dbObj.	tst_dbobj.c, tst_insert2.c
gdi_obj_container_destroy	Destroys a tuple container.	
gdi_obj_tuple_create	Creates a tuple.	tst_dbobj.c, tst_insert2.c
gdi_obj_tuple_destroy	Destroys a tuple.	tst_dbobj.c, tst_insert2.c
gdi_obj_tuple_add	Adds a tuple to a tuple container.	tst_dbobj.c, tst_insert2.c
gdi_obj_tuple_retrieve	Retrieves a tuple from a tuple container.	tst_constr.c, tst_dbobj.c, tst_insert2.c
gdi_obj_fill_data	Inserts data into a tuple.	tst_dbobj.c, tst_insert2.c
gdi_obj_get_data	Reads data from a tuple.	tst_constr.c, tst_dbobj.c, tst_insert2.c

5.2 Column Definitions

The dbObj stores information about each column in an array of dbColDef structures, defined in *libgendb.h* and depicted in Figure 9.

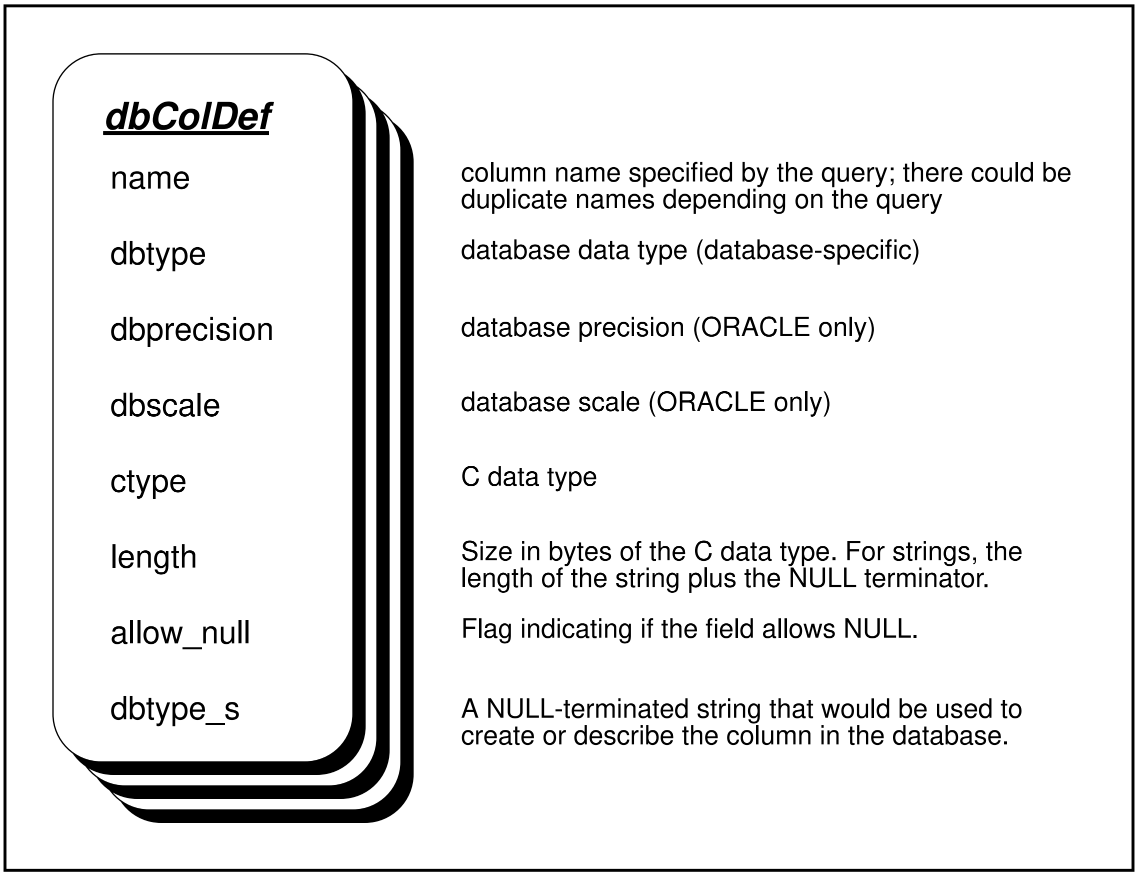

FIGURE 9. dbColDef Structure

Like the dbObj, the dbColDef should not be accessed directly. Instead the functions and macros listed in Table 8 should be used.

Table 9. Summary of dbColDef Macros and Functions

Name	*Description*	*Sample Code*
gdi_col_def_create	creates a new column definition	tst_create.c, tst_dbobj.c, tst_insert2.c
gdi_col_def_destroy	destroys (deallocates) a column definition.	
gdi_col_def_add	Adds a column definition created with *gdi_col_def_create()* to a dbObj.	tst_create.c, tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_NAME	Get the name of a column given a column number.	tst_dbobj.c, tst_insert2.c

Table 9. Summary of dbColDef Macros and Functions

Name	Description	Sample Code
GDI_OBJ_COL_CTYPE	Get the C type of a column given a column number.	tst_constr.c, tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_PRECISION	Get the precision of a column given a column number.	tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_SCALE	Get the scale of the column given its column number.	tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_LENGTH	Get the length of the column.	tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_DBTYPE	Get the database data type for a column.	tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_DBTYPE_S	Get the database string for creating or describing a column.	tst_dbobj.c, tst_insert2.c
GDI_OBJ_COL_ALLOW_NULL	Get the allow_null flag.	tst_dbobj.c, tst_insert2.c

5.3 Tuple Constructor

The tuple constructor is specified at the time a dbObj is created. It stores pointers to the routines that are actually invoked when the user application calls subsequent GDI routines, thus hiding lower level data structures.

For example, when an application calls *gdi_obj_get_data(), gdi_def_get_data()* is actually invoked if the dbObj was created with GDI_DEFAULT, and *gdi_sdi_get_data()* is invoked if the dbObj was created with GDI_SDI_CONSTR.

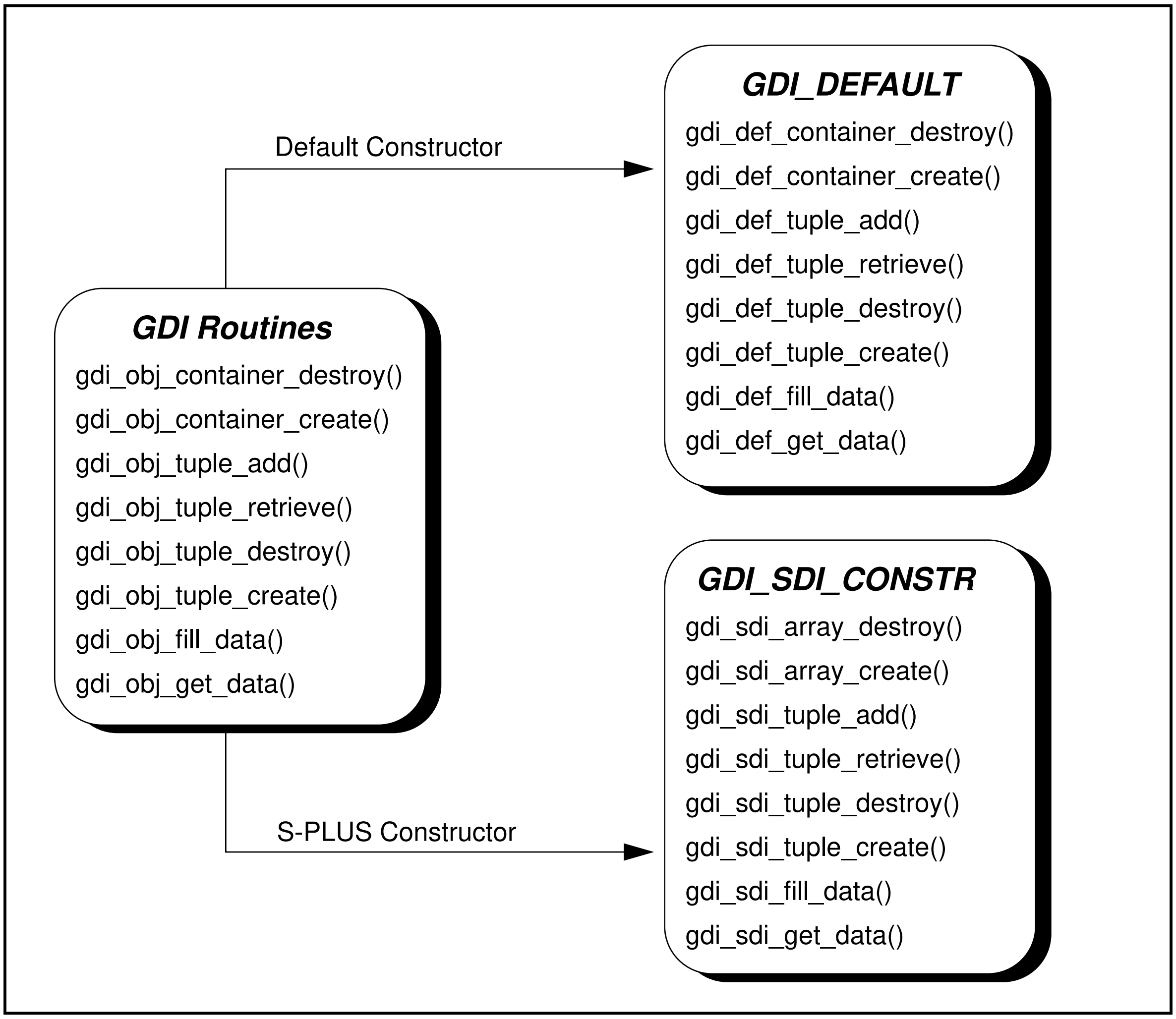

FIGURE 10. Tuple Constructor

6. Error Handling

Errors are managed on a connector by connector basis, each dbConn storing information for activity on its channels. The status of a function, whether it succeeds or fails (GDI_SUCCESS or GDI_FAILURE), is always recorded in the dbConn along with the specific error code and message string. The dbConn stores information about the last command executed, overwriting previous statuses. For that reason, the dbObj also records the exit status.

Some functions, such as dbObj functions, do not have a dbConn. Also, an application does not have a dbConn until a call to *gdi_open()* succeeds. For these cases, the error code and text are stored in a global location accessed by specifying a NULL dbConn.

Figure 11 depicts how an error that may have occurred inside a GDI subroutine gets communicated back to the user.

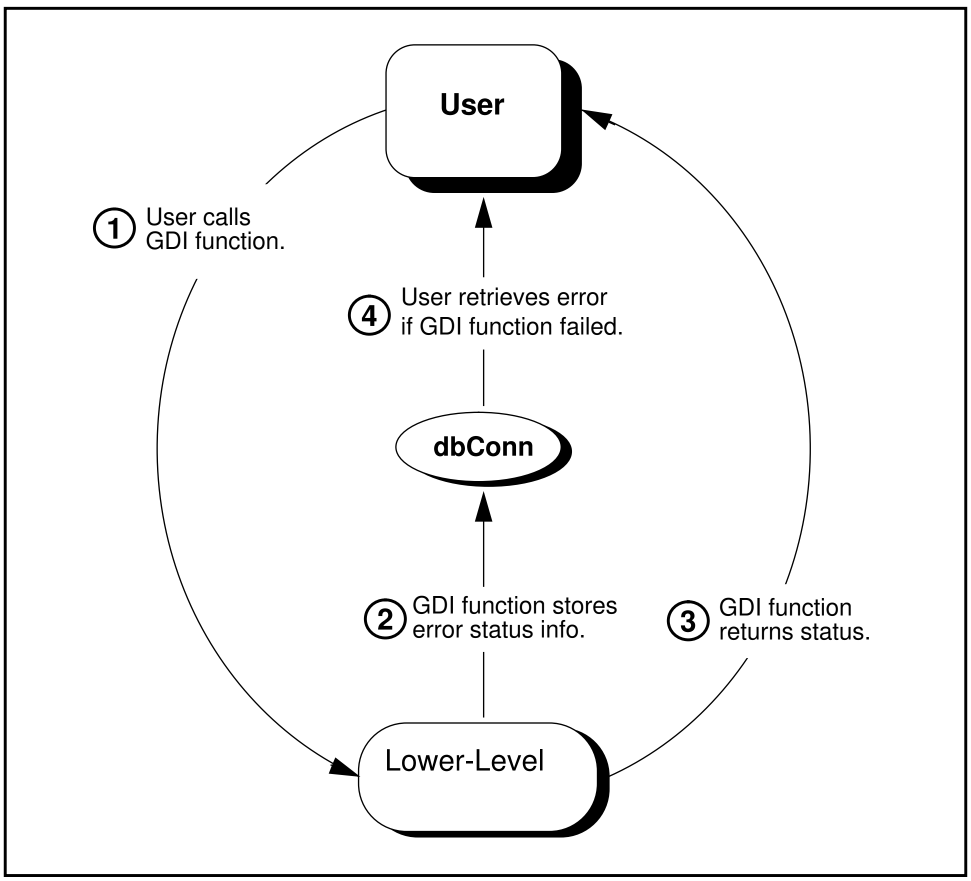

FIGURE 11. GDI Error Handling

Two sets of error handling functions, one for the user and one for the lower-level GDI functions, provide error handling capabilities and are described in the following two sections.

6.1 User Error Functions

This section discusses what the user must know to manage errors, including how to:

- Detect if a GDI function failed.
- Retrieve the error from the dbConn.
- Control whether database warnings return GDI_SUCCESS or GDI_FAILURE.
- Debug problems.

A user detects failure by checking the return status of a function. Most GDI functions return GDI_SUCCESS or GDI_FAILURE. Information about the error is stored in the dbConn used in the function call. For example:

Example 14:

```
if (gdi_commit (my_dbConn, channo) != GDI_SUCCESS)
{
            gdi_error_get (my_dbConn, &errcode, errtext, maxtextlen, &status, &severity);
            fprintf (stderr, "%s\n", errtext);
}
```

Functions that allocate structures, such as *gdi_open()*, return a pointer to the new dbConn structure. A NULL return pointer indicates that the routine has failed. The following *gdi_open()* call demonstrates both how to check for a NULL return and how to retrieve an error from the NULL dbConn:

Example 15:

```
if ((my_dbConn = gdi_open (vendor, account, password, database, server, appname))
      == (dbConn *) NULL)
{
            gdi_error_get ((dbConn *) NULL, &errcode, errtext, maxtextlen, &status, &severity);
            fprintf (stderr, "%s\n", errtext);
}
```

Sometimes a database generates a warning which may or may not be important to an application. For instance, ORACLE databases set a warning flag under the following conditions:

- A user updates or deletes a table without a where clause.
- A fetch truncates data in a column.

The user can instruct the GDI to treat such warnings as fatal by setting the *gdi_error_init()* argument, *threshold*, to GDI_WARNING. The *threshold* indicates the error level that is considered a failure and which cause a GDI function to return GDI_FAILURE. The *threshold* may be changed at any time and the current setting may be checked with a call to *gdi_error_flags()*.

gdi_error_init() also has a *debug* flag. When set to GDI_DEBUG_ON, errors are automatically output to *stderr*. When set to GDI_DEBUG_VERBOSE, additional debug messages are automatically output to *stderr*. These options are especially useful during the early stages of application development, but should not be used as a replacement for actual error handling.

Table 10. summarizes user error handling functions and macros.

Table 10. User Error Handling Functions and Macros

Name	*Description*	*Man Page*
gdi_error_init	Optional routine that sets *debug* and the severity *threshold* level. *debug:* default setting is GDI_DEBUG_OFF. GDI_DEBUG_ON outputs errors to *stderr*. GDI_DEBUG_VERBOSE outputs any additional debug messages to *stderr*. *threshold*: The default is GDI_WARNING, which means that GDI_SUCCESS is returned if a warning occurs. If set to GDI_FATAL, then warnings return GDI_FAILURE.	gdi_error_init(3)
gdi_error_get	Retrieves error code, error text, severity, and exit status from the dbConn.	gdi_error_get(3)
gdi_error_flags	Retrieves the current setting of *debug* and *threshold* from the dbConn.	gdi_error_flags(3)
gdi_trace	Flips vendor specific database tracing on or off.	none yet
GDI_OBJ_STATUS	The exit status in the dbObj (GDI_SUCCESS or GDI_FAILURE).	

6.2 Lower-Level Error Functions

The lower-level routines, summarized in Table 10., store errors in the dbConn. These functions should not be called by user applications. Developers writing GDI functions that will be called by user applications should be aware of these functions.

Table 11. Lower-Level Error Setting Functions

Name	*Description*	*Man Page*
gdi_error_app	Sets error code and text in the dbConn.	
gdi_warning_app	Sets a GDI warning. If the threshold is set to higher than GDI_WARNING or if the error If code is GDI_NOERROR then the dbConn status is set to GDI_SUCCESS. Otherwise the status is set to GDI_FAILURE.	
gdi_error_unix	Gets error code from Unix *errno* and error text from *syserrorlist* If a UNIX error occurred (for example, a *malloc* failed). Stores in dbConn by calling *gdi_error_app*().	
ora_sqlca_error	ORACLE-specific routine that stores SQLCA error information in the dbObj. For use by PRO*C routines.	ora_sqlca_error(3)

6.3 Known Problems

Asynchronous Processing

Since errors are managed at the dbConn level, channels that execute commands asynchronously should not belong to the same dbConn since they will overwrite each other's error status. In this case, additional dbConn structures should be used.

ORACLE and Signals

ORACLE is signal-sensitive, using SIGINT for its network communications. Special ORACLE-provided routines must be used to put alternate SIGINT handlers in place. For more information, see your local ORACLE Database Administrator.

POSTGRES

Be aware that POSTGRES error-handling in the current baseline release is weak and is being addressed in the next release.

7. Transaction Management

A transaction is a group of database statements that are treated as a single unit, *i.e.*, the effects are seen in their entirety or not at all. If queries executed inside a transaction change the database, those changes do not become permanent until the transaction is committed. A *rollback* negates all changes.

Each database manages transactions differently. By default, each POSTGRES and SYBASE statement commits as soon as it has successfully completed; you must explicitly begin a transaction to group multiple statements together. *gdi_begin_tran()* starts a transaction for POSTGRES and SYBASE databases. No changes will become permanent until a *gdi_commit()* is executed, and all changes may be undone with *gdi_rollback()*.

By default, ORACLE implicitly starts a transaction with the first database statement. No changes become permanent until a *gdi_commit()* is executed, and all changes may be undone with *gdi_rollback()*. *gdi_auto_commit()* puts ORACLE into a mode where every statement commits automatically as soon as it completes.

Two conditions may automatically cause a commit, depending on the database:

- A DDL statement, such as create or drop, commits pending changes even if the statement itself fails.
- *gdi_close()* commits pending changes before terminating the database connection.

In general, it is better to explicitly commit or rollback by storing the proper statement in a query string and executing it with *gdi_submit()* or by using one of the functions summarized in Table 12.

Table 12. Transaction Management Functions

Function	*Description*	*Database*
gdi_begin_tran	Begin a multi-statement transaction	POSTGRES, SYBASE
gdi_commit	End a transaction, making all changes permanent.	all
gdi_rollback	End a transaction, discarding all changes.	all
gdi_savepoint	Set a savepoint.	ORACLE, SYBASE
gdi_auto_commit	Have each statement automatically commit if it succeeds.	ORACLE