February 1994

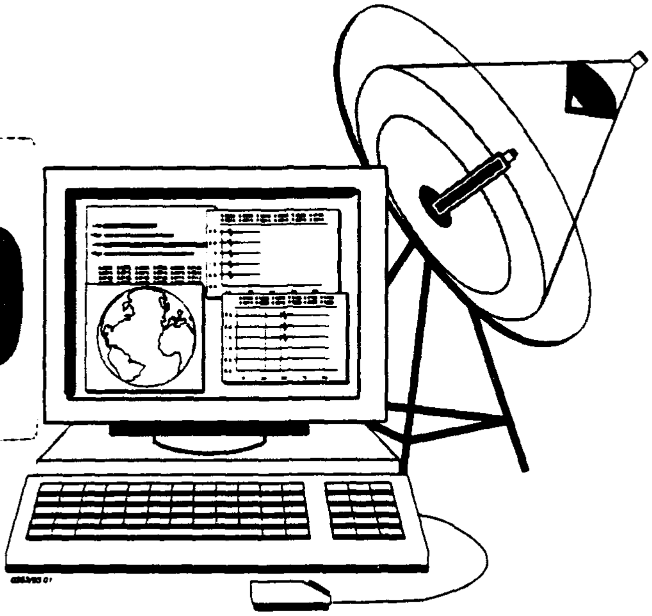
SAIC-93/1001 REV

2

The Intelligent Monitoring System

Generic Database Interface (GDI)

User Manual



SPECIAL TECHNICAL REPORT

25 February 1994

Baseline 21.1

Jean T. Anderson, Mari Mortell, Bonnie MacRitchie, Howard Turner

Geophysical Systems Operation

140Pg

SAIC

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Sponsored by:
ADVANCED RESEARCH PROJECTS AGENCY
Nuclear Monitoring Research Office
ARPA Order Number 6266, Program Code No. 62714E
Issued by: DARPA/CMO
Contract No. MDA972-92-C-0026

Principal Investigator:
Dr. Thomas C. Bache
(619) 458-2531

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
3-Jan-94

3. REPORT TYPE AND DATES COVERED
Special Technical 11/27/91-2/25/94

4. TITLE AND SUBTITLE
The Intelligent Monitoring System
Generic Database Interface (GDI) User Manual

5. FUNDING NUMBERS
MDA972-92-C-0026

6. AUTHOR(S)
Jean Anderson, Mari Mortell, Bonnie MacRitchie,
Howard Turner

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Geophysical Systems Operation
Science Applications International Corporation
10260 Campus Pt. Drive
San Diego, CA 92121

8. PERFORMING ORGANIZATION REPORT NUMBER
SAIC-93-1001REV

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Advanced Research Project Agency
3701 N. Fairfax Drive, #717
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Generic Database Interface (GDI) is a common application programmable to multiple databases, providing two key capabilities: Database access and data management. Database access routines allow an application to connect to and query a database with the same GDI call whether the target database is ORACLE, POSTGRES, or SYBASE. Data to and from the database are managed in the native format of the application, making it possible to provide a seamless integration of application and database.

14. SUBJECT TERMS
Data Management, Relational Database, Generic Interface

15. NUMBER OF PAGES
140

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
NONE

Table of Contents

Part I: Introduction

Part II: Generic Interface

Part III: High-Level Interfaces

Part IV: Reference Manual

Utilities:

Generic API:

ORACLE-Specific Routines:

FORTRAN API:

Part V: Appendices

Part I: Introduction

1. Overview

The Generic Database Interface (GDI) is a common Application Programming Interface (API) to multiple databases. The GDI provides two key capabilities:

1. *Database Access*
   An application connects to a database and executes a database query with the same GDI calls whether the target database is ORACLE, POSTGRES, or SYBASE.
2. *Data Management*
   Data to and from the database are managed in the native format of the application, making it possible to provide a seamless integration of application and database.

The GDI model consists of the components depicted in Figure 1. High-Level interfaces may be added without having to modify lower level functionality.

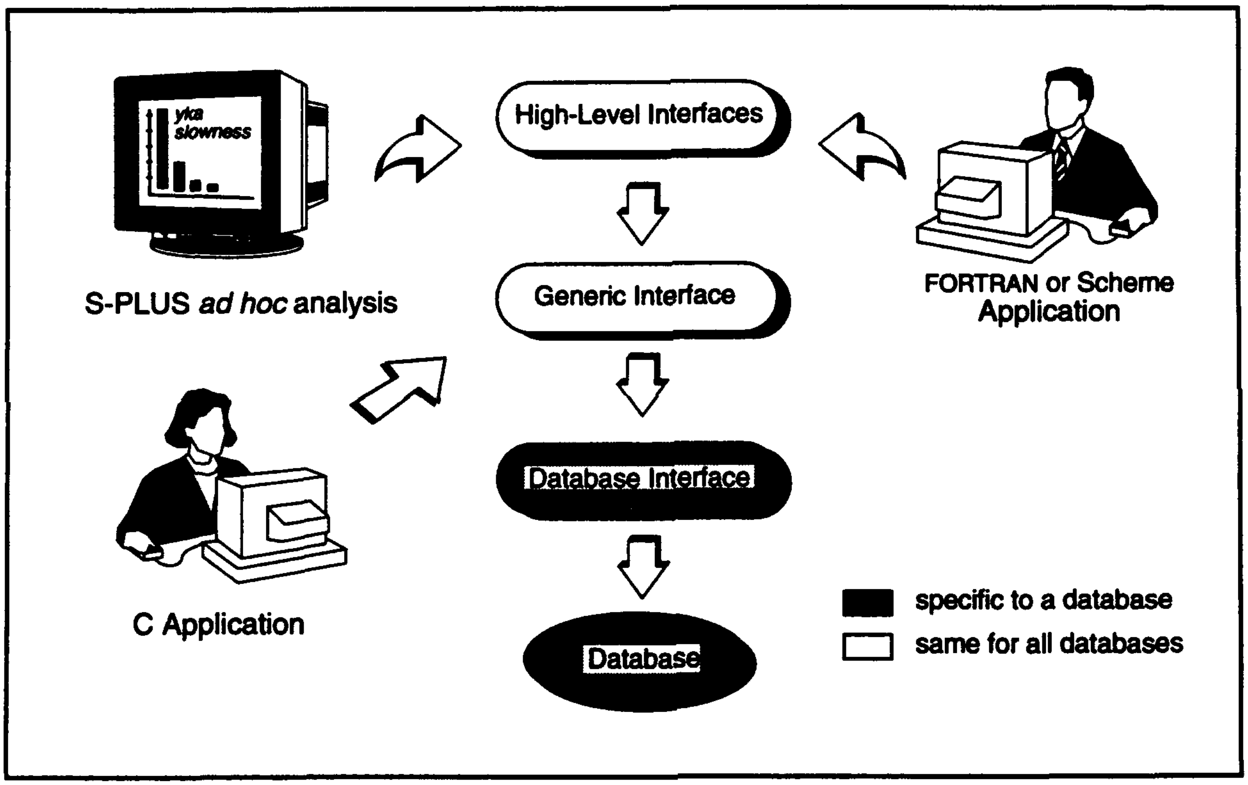


FIGURE 1. Generic Database Interface (GDI) Model

Working from the bottom of Figure 1 to the top, the GDI consists of:

- *Database Interface*
  Manages interaction with the target database.
- *Generic Interface*
  Provides a common API for C applications to access any database and manage data.
- *High-Level Interfaces*
  Support programming languages such as FORTRAN and Scheme, and third-party products such as S-PLUS.[1]

1.1 Intended Audience

The GDI targets two types of users: the end-user and the application developer. Section 10 describes S-PLUS, an end-user application.

The end-user interactively accesses the database with a program created by an application developer or a third party tool such as S-PLUS. End-users want a "hot link" between the application and the target database so they can concentrate on research and analysis. They do not want to be sidetracked by having to manually transfer data, not even with the aid of data migration tools.

The application developer writes programs that require database access. Application developers want a consistent interface between the application and the target database so they can concentrate on a specific area of programming expertise, whether it be the design of sophisticated user interfaces or complex scientific programs. They do not want to be sidetracked by having to learn how to access each database.

Neither user wants to become an expert for each database accessed. Both want application and database to be transparently integrated. The GDI achieves that transparent integration.

This manual describes what each user must know to submit queries to a database and manage data. The user needs to know:

- The database query language, which is a topic beyond the scope of this document. Appendix A lists a few SQL references. POSTGRES documentation is available *via* anonymous ftp from *postgres.berkeley.edu.*
- How to use the generic functions that execute queries and manage data. This is the topic of this manual.

The user does not need to know:

- Database vendor-specific implementation of Embedded SQL and/or the call interface.
- Database vendor-specific data dictionary structure.
- Database vendor-specific error handling.
- Application-specific and database-specific data formats.
- Internal GDI data structures.

1.2 Document Organization

PART I introduces a high-level view of the GDI. Section 1 (this section), describes the GDI model. Section 2 describes the GDI architecture.

PART II introduces GDI routines to the application developer. Section 3 discusses naming conventions, sample programs, and known problems. Section 4 discusses database communications. Section 5 and Section 6 describe query execution and specialized database functions.

[1] S-PLUS is a statistical and graphics program developed by StatSci that is based on the S-Language.

Section 7 describes data management. Section 8 and Section 9 discuss error handling and transaction management, respectively.

PART III introduces the high-level interfaces to the end-user. Section 10 contains an S-PLUS tutorial. Section 11 describes the FORTRAN interface.

PART IV contains UNIX Section 1 man pages for GDI tools and Section 3 man pages for GDI routines. The most current man pages are available on-line.

PART V contains appendices. Appendix A is a bibliography of SQL language references. Appendix B is a description of GDI data types.

1.3 User Feedback

The GDI development team welcomes comments. All bug reports and suggestions for improvement should be sent to *gdi@gso.saic.com.*

2. Architecture

Section 1 presented a high level view of the GDI. This section describes the key components of the GDI architecture:

- *Basic Services*: Database access routines.
- *Database Connector (dbConn):* Manages database queries.
- *Database Object (dbObj):* Manages data to and from the database.

Figure 2 depicts how an application uses the dbConn and dbObj to access a database. All queries are executed on the dbConn that was established when the application connected to the database. This is similar to a C program using a FILE pointer for reads and writes to a file opened with *fopen()*. If a query returns data, the GDI returns a pointer to the dbObj containing the data. If an application needs to insert data into the database, it can create a dbObj and populate it with the data to be inserted.

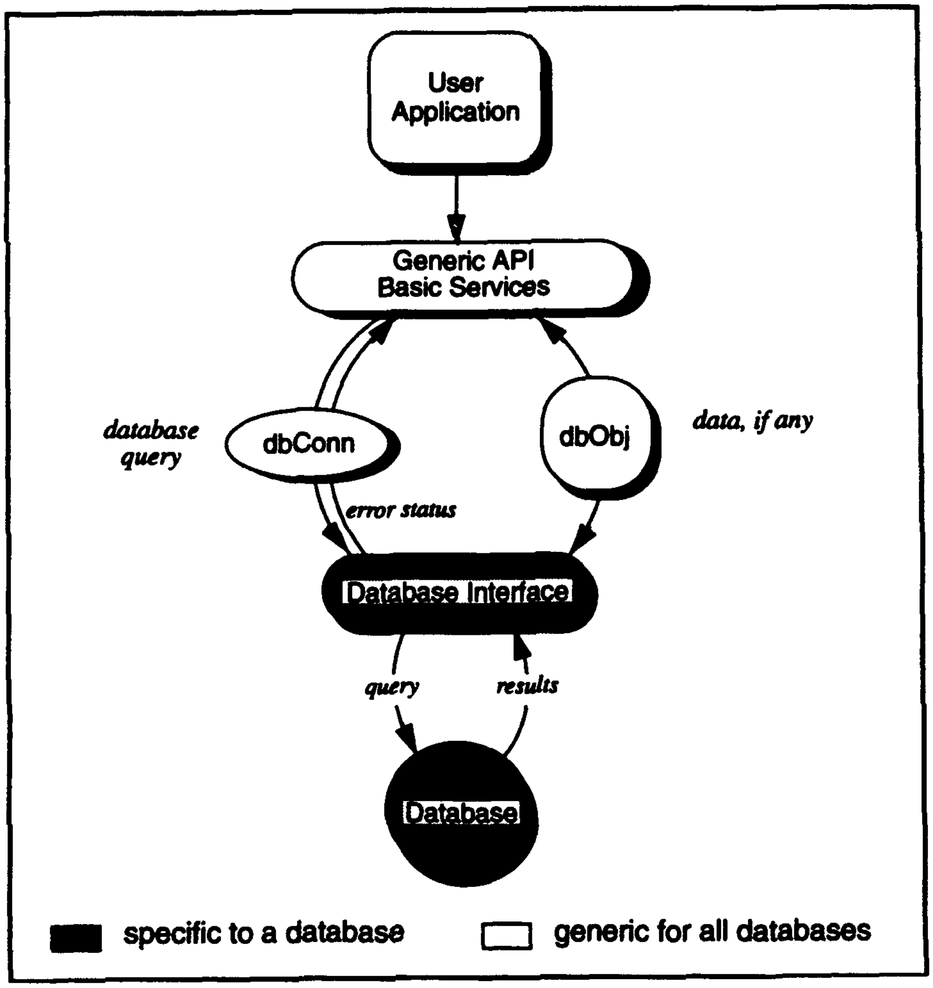


FIGURE 2. Generic Database Interface Architecture

2.1 Basic Services

GDI routines are organized into the following areas that provide:

1. *Communications*
   Database opens and closes, query cancellation, and query tracing.
2. *Error Handling*
   Consistent error reporting whether the actual error was a database error, a UNIX error, or an application-specific error. The application can decide whether warnings should be treated as fatal and a debug option automatically outputs errors to *stderr* to aid developers in debugging problems.
3. *Transaction Management*
   Hooks for starting a multi-statement transaction (POSTGRES and SYBASE), and for issuing commits, rollbacks, and savepoints.
4. *Data Dictionary Access*
   Consistent interface to each vendor's data dictionary for commonly asked questions such as "what is this object?", "what is its structure?", "who owns it?"
5. *Canned Database Queries*
   Highly optimized database access for commonly required functionality. For example, some vendor products have sequencing mechanisms while others do not. The *gdi_get_-counter()* routine provides a highly optimized, consistent mechanism for fetching unique id's regardless of database.
6. *Dynamic Queries*
   Support for dynamic queries.
7. *Data Management*
   Data are managed in native application data format.

2.2 Database Connector (dbConn)

The Database Connector (dbConn) manages queries. When an application connects to the database, the GDI creates a dbConn that keeps track of administrative information, such as:

- database vendor type (*i.e.,* ORACLE, POSTGRES, SYBASE)
- database name, account, and node
- error information for the last query executed (specific database error code and string)

A single application can have multiple dbConn's, consisting of multiple connections to the same database or to a mixture of databases, as depicted in Figure 3.[1]

[1] Only one connection to POSTGRES is allowed at this time, but an application may mix one POSTGRES connection with many ORACLE and SYBASE connections.

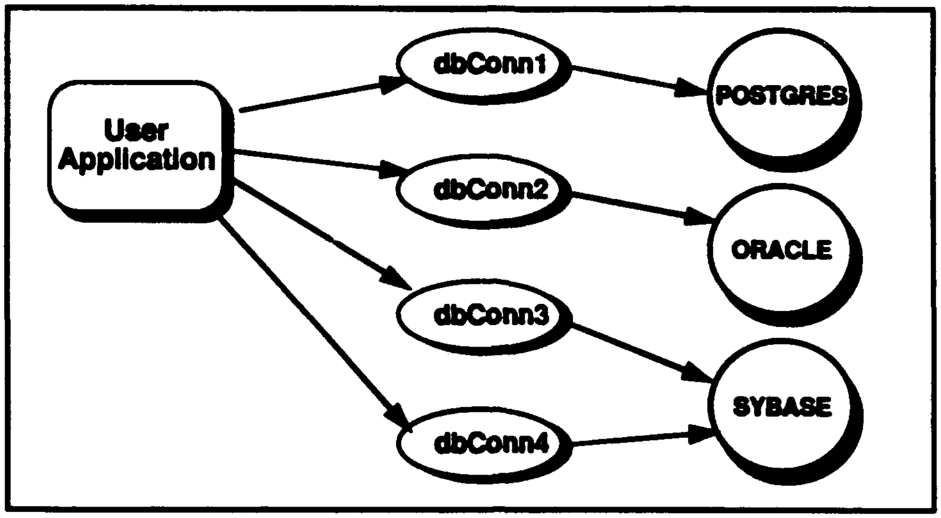


FIGURE 3. Database Connector (dbConn)

The dbConn also keeps track of the query *channel*, a communications "pipe" on which database queries are managed and executed. A channel is a DBPROCESS for SYBASE, a cursor for ORACLE, and a portal for POSTGRES. Each dbConn is initialized with at least one channel for default query activity, but users may add as many channels as they like, as depicted in Figure 4.

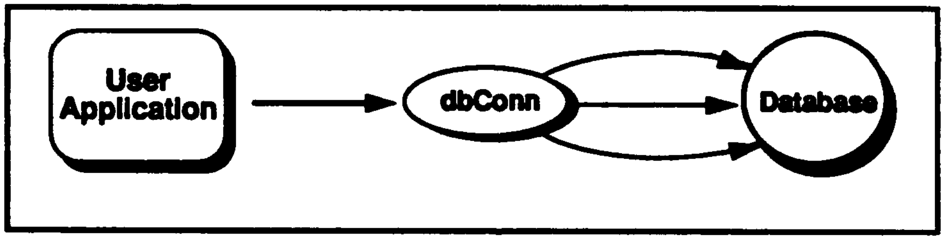


FIGURE 4. Database Query Channels

2.3 Database Object (dbObj)

The Database Object (dbObj), depicted in Figure 5, manages data and is composed of the following internal structures:

- *Tuple Container*
  Stores the data, which might be query results from a SELECT (outputs), or data to be inserted into the database (inputs). By default, data are organized into rows and columns, like a database table. The exact organization is controlled by the Tuple Constructor.
- *Column Definitions*
  Describes each column in the tuple container, including name, data type, and length.
- *Tuple Constructor*
  Specifies how to manage data in the tuple container. For example, S-PLUS operates on columns and rows instead of on rows and columns. The S-PLUS custom interface,

described in Section 10, uses an S-PLUS tuple constructor instead of the default tuple constructor. While the specific data format is intended to be transparent to the end-user, Section 7.3 describes how programmers may create tuple constructors to fit a particular application need.

- *Query Information*
  Retains query information, such as the database query string, whether or not the query succeeded, and how many rows were affected. The dbObj retains general GDI information with each result set, while the dbConn stores specific database error information about the last query executed.

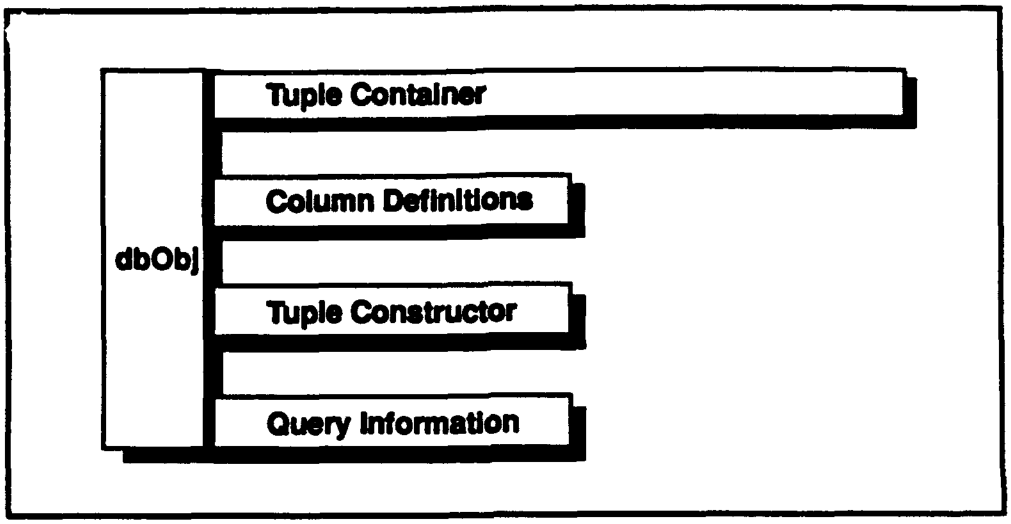


FIGURE 5. Database Object (dbObj)

The GDI provides functions and macros for accessing a dbObj. The user does not need to know the internal structure.

2.4 Comparison to Previous Interfaces

SAIC has developed several database library interfaces. They supported the most basic database services, the first five items discussed in Section 2.1. But none of them supported fully dynamic queries and data management, resulting in two fundamental flaws:

- Libraries were Schema-Driven.
- Data structures were inflexible.

This section describes how the dbObj solved both these problems.

Schema-Driven Libraries

Fully dynamic database selects were difficult to support because there was not a straight-forward way to pass dynamic query results back to the calling application. Instead, insert and fetch routines, with the corresponding C and FORTRAN program headers, were generated automatically for each table based on its definition in the database. If the structure of the database changed, the push of a button would regenerate the support library.

In essence, the database access library was hard-coded to the schema being accessed, an approach that had serious limitations:

- *Poor Support for New or Changing Database Structures*
  Applications could not access newly created tables until headers and routines had been generated, the library remade and reinstalled, and the application recompiled. Modifying existing tables required synchronizing changes to database tables, access libraries, program headers, and the applications. The library became a weak link between the application and the database.
- *Inflexible SELECT Lists*
  Since the SELECT list was hard-coded to a single table, an application received all fields in a table even if it wanted just one. More importantly, an application queried one table at a time, even though it might need data from many tables. The application had to select from each table separately, then merge the results. Because of this, the number of application-specific routines grew, defeating one of the primary purposes of a centralized library which is to reuse code.

The dbObj overcomes the problem of managing dynamically defined query results. Applications may access new tables as soon they are created, access existing tables as they are changed, and execute any database statement that is legal for the target database.

Inflexible Data Structures

Previous interfaces supported one data structure: an array of structures. If an application needed a linked list, it constructed the list and copied the data into it. Likewise, data were copied to FORTRAN storage. Loading data into S-PLUS required dumping results to a flat file, then manually describing and loading the file into S-PLUS. Too many steps were required to migrate or copy data into the application.

The dbObj reduces data copying by supporting the application structure directly.

2.5 Restrictions

While an application may attach to multiple databases simultaneously, no effort is made to translate queries for the target database; the GDI passes the query straight through.

SQL Support

Commercial relational databases extend the ANSI SQL standard with features that are not guaranteed to work with other products. For example, a query containing the ORACLE outer join operator (+) will fail if it is sent to a SYBASE database which uses the asterisks (*) as the outer join operator.

The GDI passes database queries directly to the database. It does not parse nor translate queries to another vendor's SQL dialect. Vendor-specific features should be avoided. Appendix A notes which references describe ANSI SQL.

Transaction Management

Transaction management and query channels are handled differently by the various database vendors. Some functions are only applicable to a subset of the supported databases. Other functions have different effects depending on the target database.

Part II: Generic Interface

3. Introduction

This part of the GDI User Manual describes the functions that provide the following capabilities to an application developer: The application developer must know C and SQL.

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

3.1 Location of GDI Components

Table 1 summarizes the location of GDI components. *INSTALL* 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*
User Manual	FrameMaker[1] source organized into a book named *gdi.bk*. A Postscript version is named *gdi.ps*.	*LIBSRC*/libgendb/doc/fm/user_manual
man pages	UNIX man pages describe each GDI function call.	*INSTALL*/man
libgdi.a, libgdiora.a	GDI libraries linked in by an application	*INSTALL*/lib
libgdi.h, gdi_f77.h	Public GDI headers that applications include in source code files.	*INSTALL*/include
gdi_gen_Astructs	Header generator for ArrayStructs tuple constructor; see gdi_gen_Astructs(1).	*INSTALL*/bin
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
source code	GDI functions.	*LIBSRC*/libgendb/src

[1] Framemaker is a document publishing tool from Frame Technology Corporation

3.2 Sample Programs

The programs in *LIBSRC*/libgendb/test exercise GDI functions and constitute sample code that demonstrate how to use the GDI. 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.

The programs use *libpar.a*, a public domain library from Caltech, to parse command line arguments. The command line arguments can be included in a parameter file (e.g. par file) and the name of the this file can be used on the command line. A par file for each test program resides in *LIBSRC*/libgendb/test. 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 accounts, passwords, database names and queries 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

3.3 Database-Specific Notes

3.3.1 ORACLE

3.3.1.1 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 Oracle Call Interface (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.

*3.3.1.2 Support for PRO*C Routines*

Currently, *gdi_open()* establishes database connections with OCI. This allows multiple, concurrent connections for applications using the GDI or their own OCI functions. Applications may link in their own PRO*C subroutine; 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 currently be opened at a time. However, additional OCI connections may be established with *gdi_open()*. A future enhancement will allow multiple PRO*C connections.

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)). Example 1 shows sample calling syntax.

Example 1:

```
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 demonstrates the PRO*C capability. *tst_get_dbcount* in *LIBSRC*/libgendb/test exercises the PRO*C function.

3.3.1.3 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
```

3.3.1.4 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 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.

3.3.1.5 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*.

3.3.2 MONTAGE

Basic hooks are in place.

3.3.3 POSTGRES

Basic hooks are in place.

3.3.4 SYBASE

Basic hooks are in place.

4. Database Communications (dbConn)

Table 4 summarizes the dat^base communications functions.

Table 4. Summary of Communication Functions

Name	Description	Man Page	Sample Code
gdi_init	Initialize the GDI library	gdi_init(3)	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 an Oracle PRO*C connection	oracle_open(3)	tst_get_dbcount.c
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)	

4.1 Connecting to a Database

gdi_init() initializes the GDI library. It takes two parameters:

appname Name of the executable.

gdihome Root directory of GDI installation. The GDI searches gdihome/lib for shared objects it dynamically loads.

gdi_init() should be called once by the application program before any other GDI functions are called.

Example 2:

```
gdi_init (argv[0], "/prj/shared/lib");
```

gdi_open() connects a process to a database and returns a dbConn structure. A NULL dbConn means the connect failed. Table 5 summarizes which databases use each parameter.

Table 5. gdi_open() Parameters

Parameter	*MONTAGE*	*ORACLE*	*POSTGRES*	*SYBASE*
vendor	yes	yes	yes	yes
account	optional	yes	no	yes
password	optional	optional	no	yes
database	optional	optional	optional	yes
server	optional	no	optional	yes
appname	no	no	no	yes

Example 3 shows how a program called SampleProgram might connect to an ORACLE database.

Example 3:

```
dbConn      *my_dbConn1;
char        *vendor="oracle";
char        *account="scott";
char        *password="tiger";
char        *db="t:host1: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 4 would create the same database login as Example 3.

Example 4:

```
dbConn      *my_dbConn1;
char        *vendor="oracle";
char        *account="scott/tiger@t:host1: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.

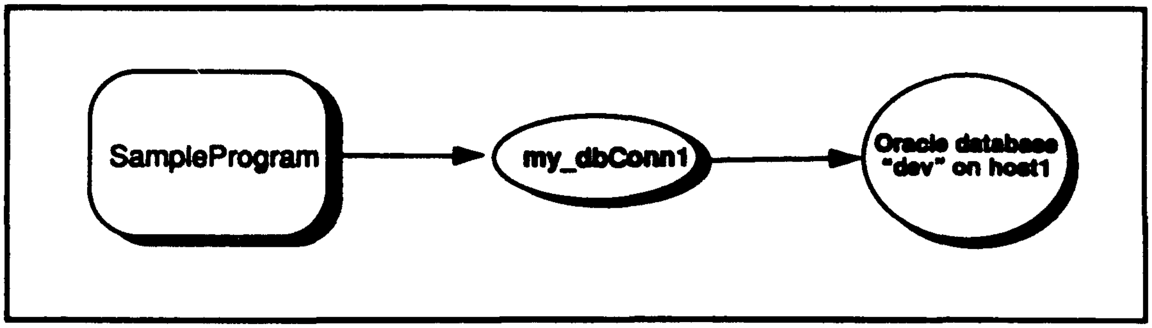


FIGURE 6. SampleProgram Connected to one Database

An application may connect to more than one database simultaneously. Example 5 shows the same process connecting to a POSTGRES database.

Example 5:

```
dbConn      *my_dbConn2;
char        *vendor="postgres";
char        *account=NULL;
char        *password=NULL
char        *db="gdidemo";
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.

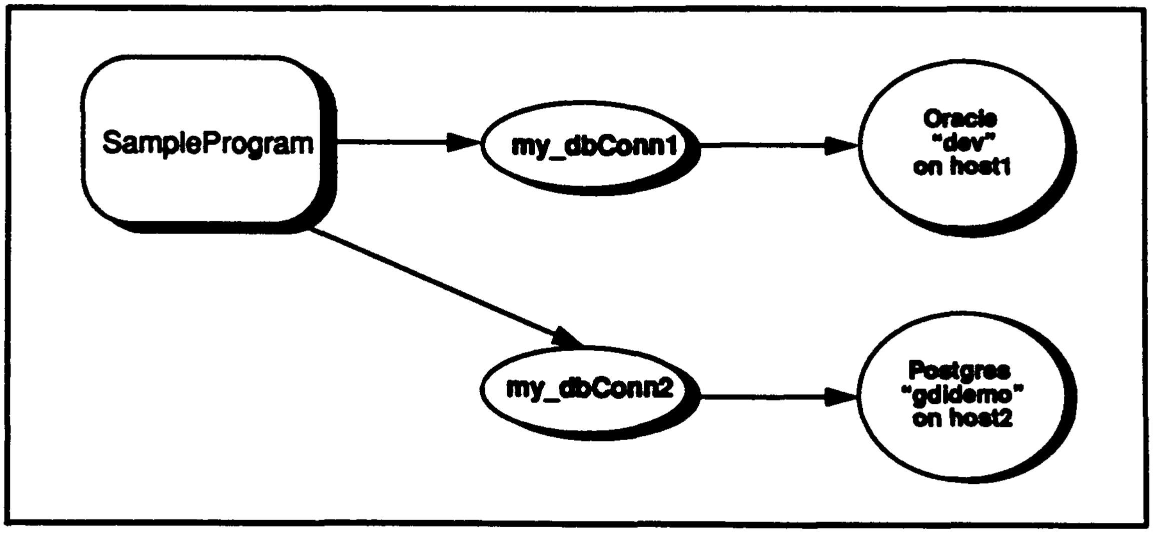


FIGURE 7. SampleProgram Connected to Two Databases

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 6 shows how the dbConn connections established by Example 4 and Example 5 could be output to *stdout*.

Example 6:

```
gdi_print_conn (my_dbConn1);
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. It is executed on a specific query channel, which is described more in Section 4.2.

Example 7:

```
if (gdi_dead (my_dbConn1, channel) == TRUE)
{
        ... connection dropped, do something appropriate ...
}
```

gdi_close() closes a specific database connection. Example 8 closes *my_dbConn*1; but *my_dbConn2* remains open.

Example 8:

```
gdi_close (my_dbConn1);
```

gdi_exit() closes all open connections. Example 9 closes both *my_dbConn*1 and *my_dbConn2*.

Example 9:

```
gdi_exit ();
```

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 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 the creation 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 an MI_CONNECTION for MONTAGE, a cursor for ORACLE, 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. *libgdi.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 10 shows how variable handling may be accommodated in an application.

Example 10:

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

If a query channel is specified for a function which operates at the connection level for that database, such as gdi_rollback() or gdi_commit(), then the channel argument will be ignored and the operation will be performed for the entire connection. This may cause confusion for applications switching between different database back-ends, such as ORACLE and SYBASE.

Example 11 creates an additional query channel. Note that the *address* of the new query channel number 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 11:

```
int     my_channel;

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

Example 12 checks to see if the channel is still open.

Example 12:

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

Example 13 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 13:

```
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 14 closes the query channel created in Example 11.

Example 14:

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

5. Query Execution

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 7.
5. Free the return results structure.

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

6. Specialized Database Functions

Table 6 summarizes the 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

7. 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 *libgdi.h* include file and depicted in Figure 8.

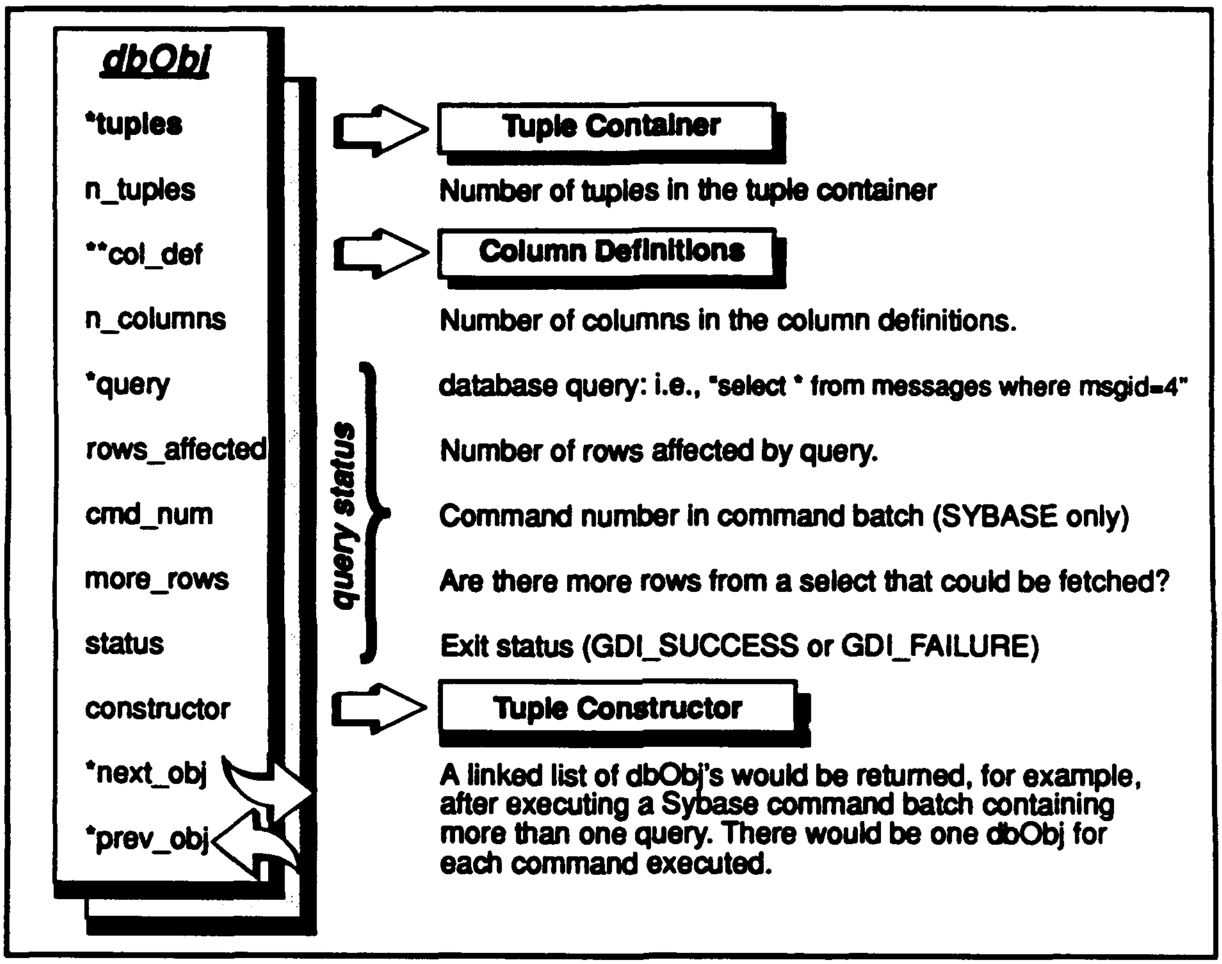

dbObj
*tuples
n_tuples
**col_def
n_columns
*query
rows_affected
cmd_num
more_rows
status
constructor
*next_obj
*prev_obj
query status
Tuple Container
Number of tuples in the tuple container
Column Definitions
Number of columns in the column definitions.
database query: i.e., "select * from messages where msgid=4"
Number of rows affected by query.
Command number in command batch (SYBASE only)
Are there more rows from a select that could be fetched?
Exit status (GDI_SUCCESS or GDI_FAILURE)
Tuple Constructor
A linked list of dbObj's would be returned, for example, after executing a Sybase command batch containing more than one query. There would be one dbObj for each command executed.

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 in Table 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 with the specified 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	
Tuple Constructor		

Table 7. Summary of dbObj Macros and Functions

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

7.1 Tuple Container

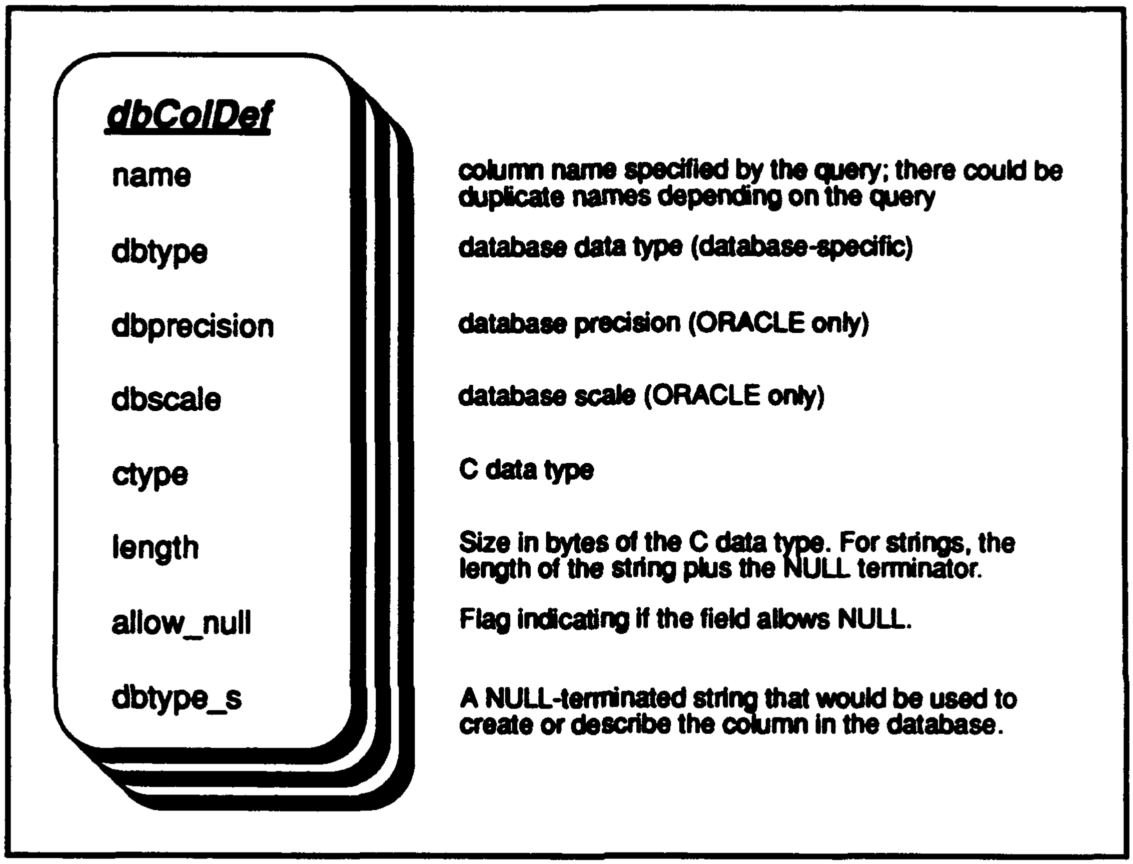
Programs do not need to know the actual structure of the tuples or of the tuple container. The functions summarized in Table 8 provide data access regardless of the actual structure.

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

7.2 Column Definitions

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



dbColDef	
name	column name specified by the query; there could be duplicate names depending on the query
dbtype	database data type (database-specific)
dbprecision	database precision (ORACLE only)
dbscale	database scale (ORACLE only)
ctype	C data type
length	Size in bytes of the C data type. For strings, the length of the string plus the NULL terminator.
allow_null	Flag indicating if the field allows NULL.
dbtype_s	A NULL-terminated string that would be used to create or describe the column in the database.

FIGURE 9. dbColDef Structure

Like the dbObj, the dbColDef should not be accessed directly. Instead the functions and macros listed in Table 9 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
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

7.3 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.

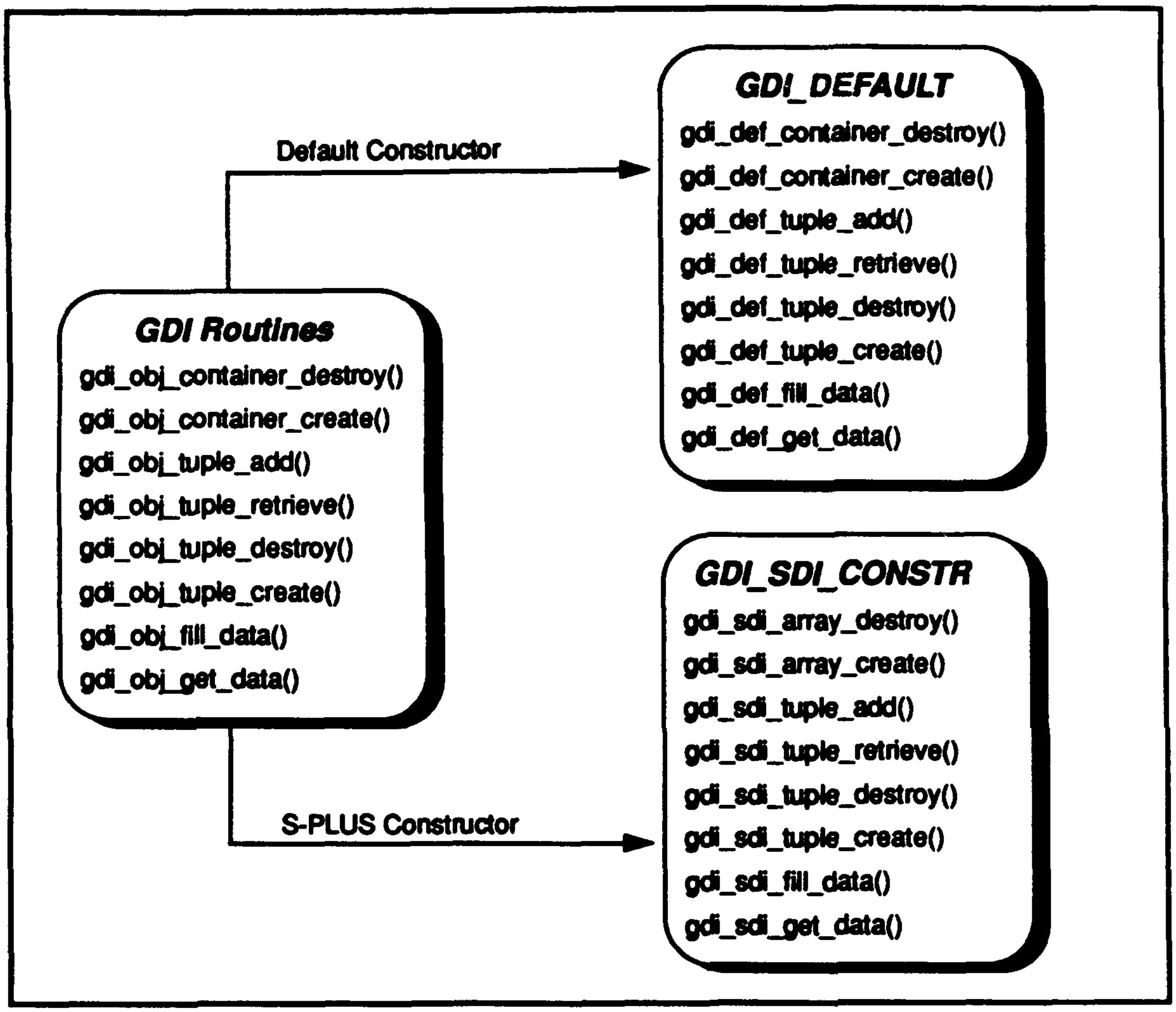


FIGURE 10. Tuple Constructor

8. 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.

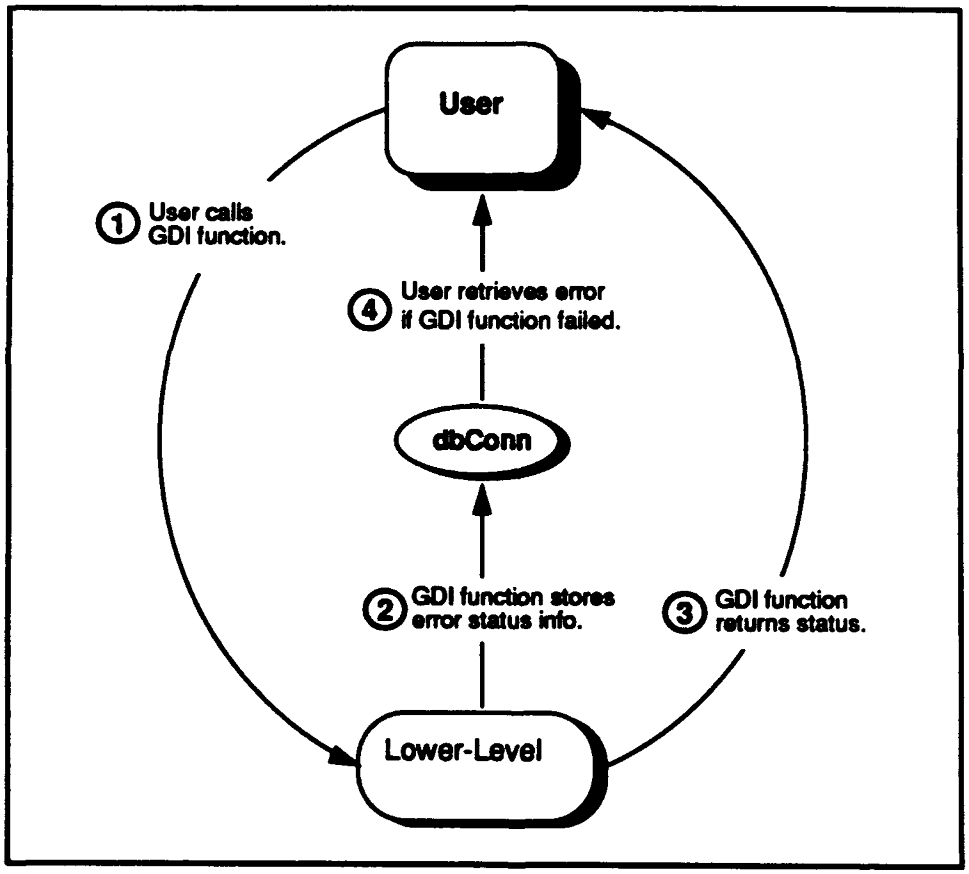

User
1 User calls GDI function.
4 User retrieves error if GDI function failed.
dbConn
2 GDI function stores error status info.
3 GDI function returns status.
Lower-Level

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.

8.1 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 15:

```
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 16:

```
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).	

8.2 Low-Level Error Functions

The low-level routines, summarized in Table 11, 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. Low-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)

8.3 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

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.

9. 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. All changes within the uncommitted transaction 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 uncommitted 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

Part III: High-Level Interfaces

10. S-PLUS Database Interface

The S-PLUS database interface lets a user interactively execute a database query at the S-PLUS prompt, then transparently transfers database query results into S-PLUS where they may be manipulated with S-PLUS functions. The databases currently supported include Montage, Oracle, Postgres, and Sybase.

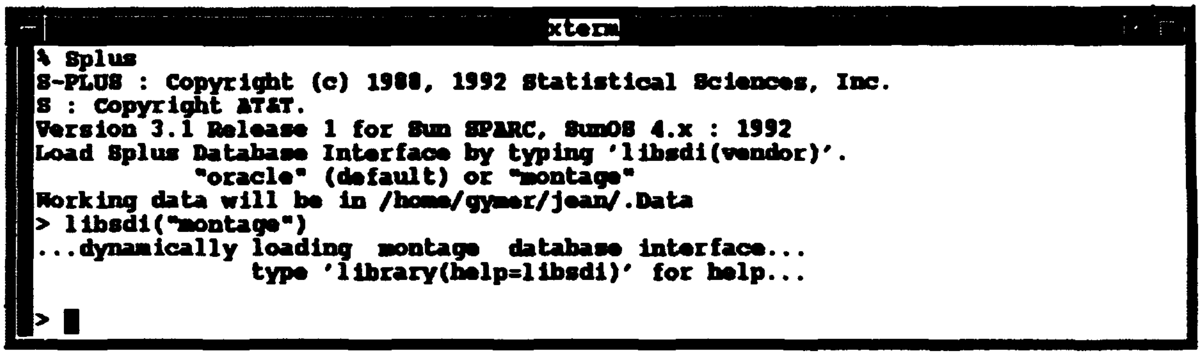
To use it, the user must know:

- The query language of the target database: SQL for Montage, Oracle and Sybase, POSTQUEL for Postgres.
- The S Language.
- How to use the following functions described in this section:

libsdi	Loads the S-PLUS Database Interface.
sdi.open	Opens a connection to a database.
sdi.submit	Executes a database query.
sdi.close	Closes the database connection.

10.1 Starting S-PLUS

Figure 12 shows how to start S-PLUS and load the database interface using the *libsdi* command, which creates the three *sdi* functions (*sdi.open, sdi.submit,* and *sdi.close*) that are used for managing a database connection and queries.



```
% Splus
S-PLUS : Copyright (c) 1988, 1992 Statistical Sciences, Inc.
S : Copyright AT&T.
Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
Load Splus Database Interface by typing 'libsdi(vendor)'.
           "oracle" (default) or "montage"
Working data will be in /home/gymer/jean/.Data
> libsdi("montage")
...dynamically loading  montage  database interface...
                 type 'library(help=libsdi)' for help...

> █
```

FIGURE 12. Loading S-PLUS Database Interface

Sites may be configured to automatically load the interface for a given database. Figure 12 is from a site that uses Oracle and Montage; Oracle is set to the default, but in this case is being overridden with the *libsdi("montage")* command.

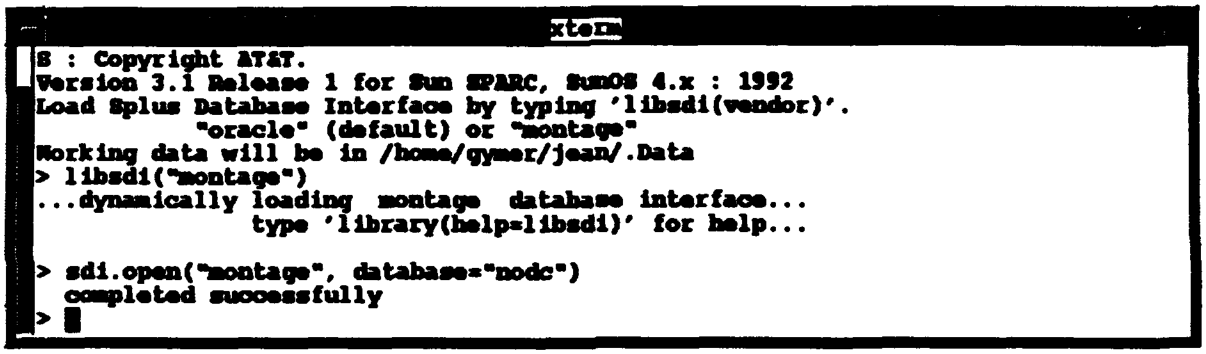
On-line help is available by entering *library(help=libsdi).*

10.2 Connecting to a Database

sdi.open() establishes a connection to the database and takes the following parameters:

vendor	Name of the database vendor (*montage*, *oracle*, *postgres*, or *sybase*).
account	Database account.
password	Password string.
database	Name of the database.
server	Database server name.
appname	Name of the application (Sybase only).

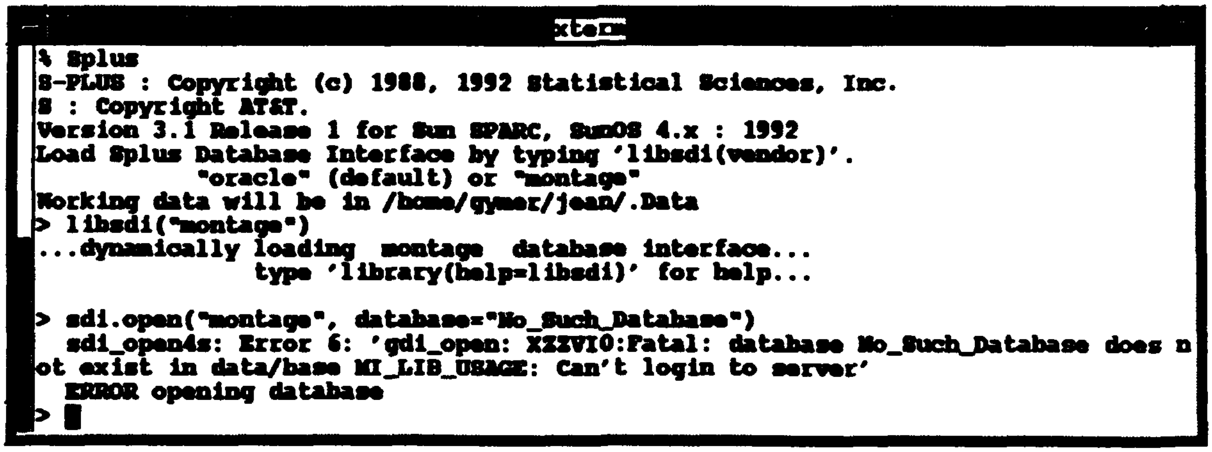
Some, or even all, of the parameters may be optional depending on the database. Figure 13 shows a user connecting to the *nodc* Montage database, using database defaults for all parameters except the database name.



```
S : Copyright AT&T.
Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
Load Splus Database Interface by typing 'libsdi(vendor)'.
            "oracle" (default) or "montage"
Working data will be in /home/gymer/jean/.Data
> libsdi("montage")
...dynamically loading  montage  database interface...
                  type 'library(help=libsdi)' for help...

> sdi.open("montage", database="nodc")
  completed successfully
> █
```

FIGURE 13. Connecting to a Database

Figure 14 shows how database errors are reported if the database connect fails.



```
% Splus
S-PLUS : Copyright (c) 1988, 1992 Statistical Sciences, Inc.
S : Copyright AT&T.
Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
Load Splus Database Interface by typing 'libsdi(vendor)'.
            "oracle" (default) or "montage"
Working data will be in /home/gymer/jean/.Data
> libsdi("montage")
...dynamically loading  montage  database interface...
                  type 'library(help=libsdi)' for help...

> sdi.open("montage", database="No_Such_Database")
  sdi_open4s: Error 6: 'gdi_open: XZZVI0:Fatal: database No_Such_Database does n
ot exist in data/base MI_LIB_USAGE: Can't login to server'
  ERROR opening database
> █
```

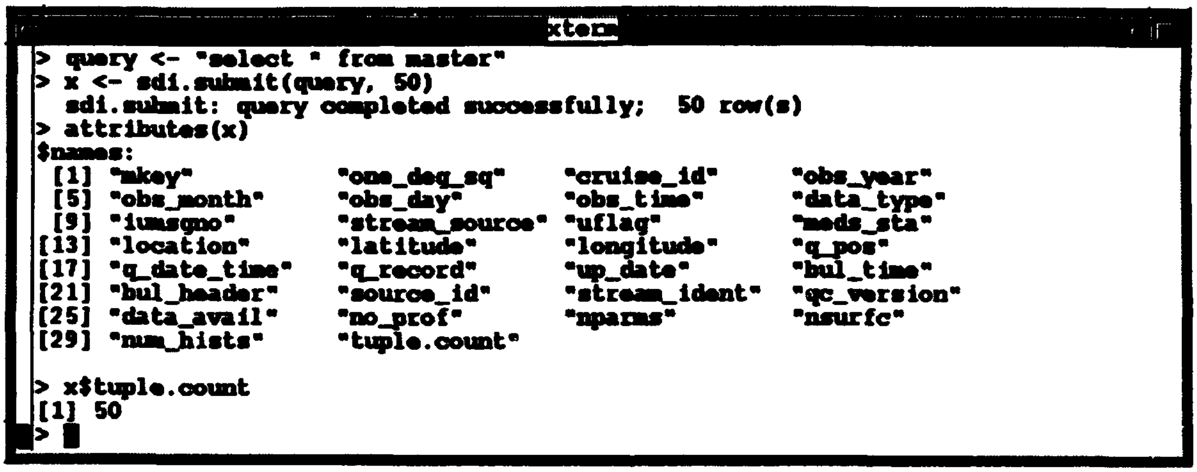
FIGURE 14. Bad Database Connection

10.3 Executing Database Queries

sdi.submit() executes database queries, taking the following parameters:

query	String containing a complete database query.
maxrec	Maximum number of records to fetch. If set to -1, all records will be returned. If set to 0, up to 500 records will be returned. Otherwise set it to the maximum number of records you want.
verbose	On by default, setting it to 0 will suppress status messages.
debug	Off by default, allows setting several debug levels to help troubleshoot any problems that might occur.

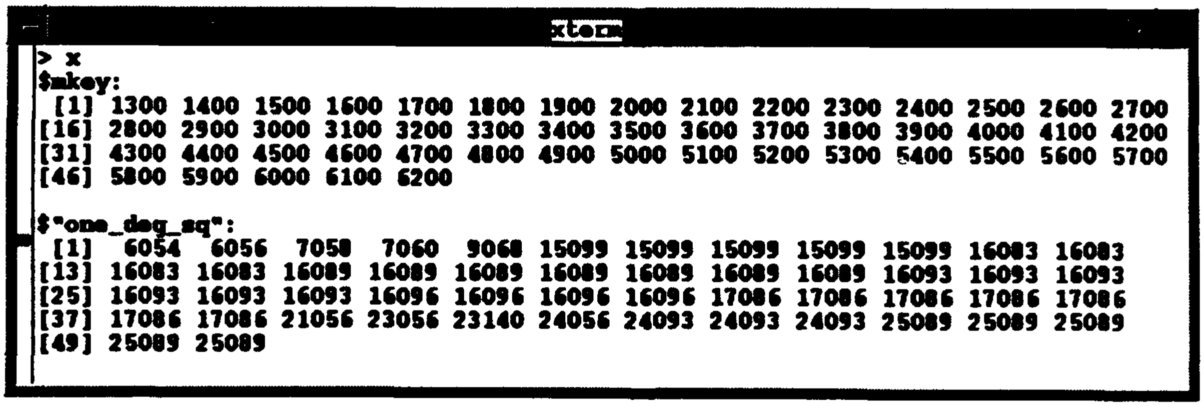
Figure 15 builds and executes a database query, requesting just the first 50 rows. It then lists the query result attributes and row count.



```
> query <- "select * from master"
> x <- sdi.submit(query, 50)
  sdi.submit: query completed successfully;  50 row(s)
> attributes(x)
$names:
 [1] "mkey"          "one_deg_sq"    "cruise_id"     "obs_year"
 [5] "obs_month"     "obs_day"       "obs_time"      "data_type"
 [9] "iumsgno"       "stream_source" "uflag"         "meds_sta"
[13] "location"      "latitude"      "longitude"     "q_pos"
[17] "q_date_time"   "q_record"      "up_date"       "bul_time"
[21] "bul_header"    "source_id"     "stream_ident"  "qc_version"
[25] "data_avail"    "no_prof"       "nparms"        "nsurfc"
[29] "num_hists"     "tuple.count"

> x$tuple.count
[1] 50
> 
```

FIGURE 15. Executing a database Query

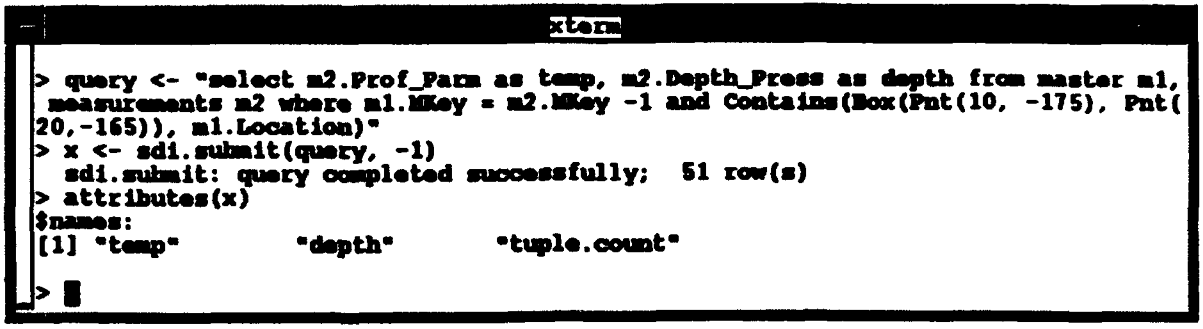
Entering *x* at the S-PLUS prompt, partially shown in Figure 16, outputs the data loaded.



```
> x
$mkey:
 [1] 1300 1400 1500 1600 1700 1800 1900 2000 2100 2200 2300 2400 2500 2600 2700
[16] 2800 2900 3000 3100 3200 3300 3400 3500 3600 3700 3800 3900 4000 4100 4200
[31] 4300 4400 4500 4600 4700 4800 4900 5000 5100 5200 5300 5400 5500 5600 5700
[46] 5800 5900 6000 6100 6200

$"one_deg_sq":
 [1]  6054  6056  7058  7060  9068 15099 15099 15099 15099 15099 16083 16083
[13] 16083 16083 16089 16089 16089 16089 16089 16089 16089 16093 16093 16093
[25] 16093 16093 16093 16096 16096 16096 16096 17086 17086 17086 17086 17086
[37] 17086 17086 21056 23056 23140 24056 24093 24093 24093 25089 25089 25089
[49] 25089 25089
```

FIGURE 16. Displaying Data

Any query legal for the target database may be executed. Figure 17 executes a more interesting query involving a join query that selects two Montage array types. In this example, it selects all available results (maxrec = -1).



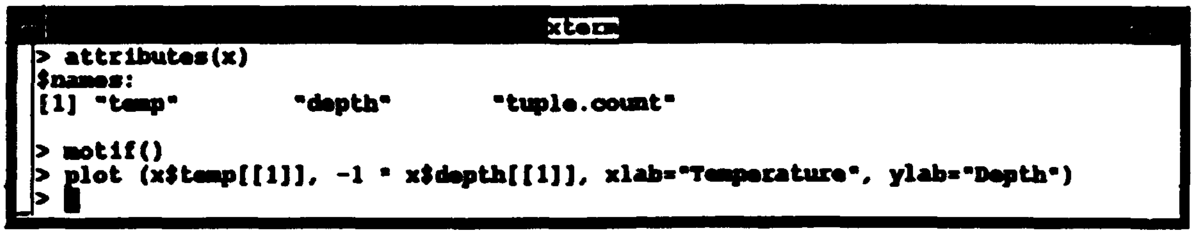
```
> query <- "select m2.Prof_Parm as temp, m2.Depth_Press as depth from master m1,
 measurements m2 where m1.MKey = m2.MKey -1 and Contains(Box(Pnt(10, -175), Pnt(
20,-165)), m1.Location)"
> x <- sdi.submit(query, -1)
  sdi.submit: query completed successfully;  51 row(s)
> attributes(x)
$names:
[1] "temp"         "depth"        "tuple.count"

> 
```

FIGURE 17. Executing a JOIN Query

While any valid query may be executed, it is important to realize that the GDI passes queries straight through to the target database. A query containing the Oracle outer join operator will fail if sent to a Sybase database and *vice versus*. Likewise, the *Contains* spatial function in the query in Figure 17 is specific to Montage and will not work if sent to Sybase or Oracle.

10.4 Plotting Results

Database query results may be manipulated with S-PLUS commands. Figure 18 creates a motif window and plots the first vector returned from the query results in Figure 17.



```
> attributes(x)
$names:
[1] "temp"        "depth"       "tuple.count"

> motif()
> plot (x$temp[[1]], -1 * x$depth[[1]], xlab="Temperature", ylab="Depth")
> 
```

FIGURE 18. Plotting Results

Figure 19 shows the results in the motif window.

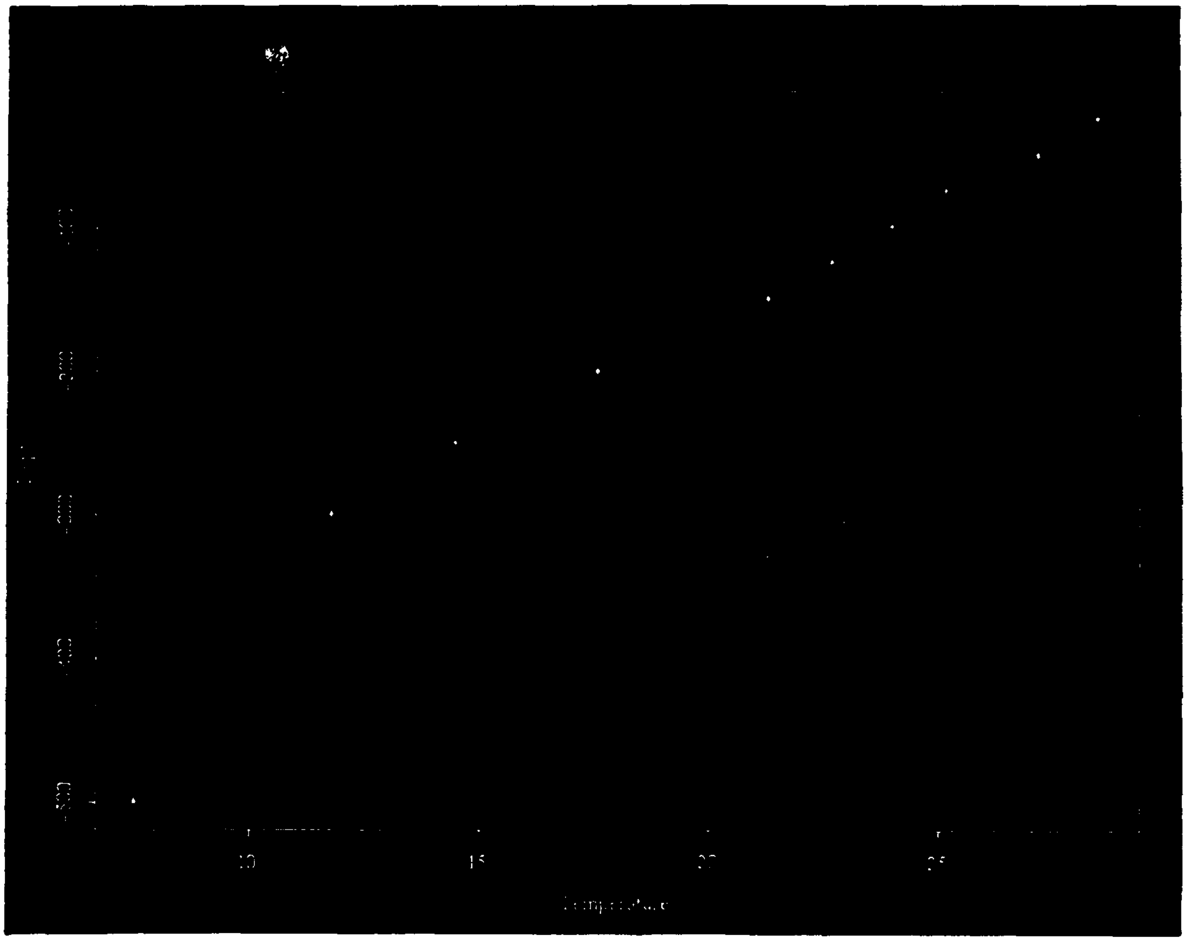
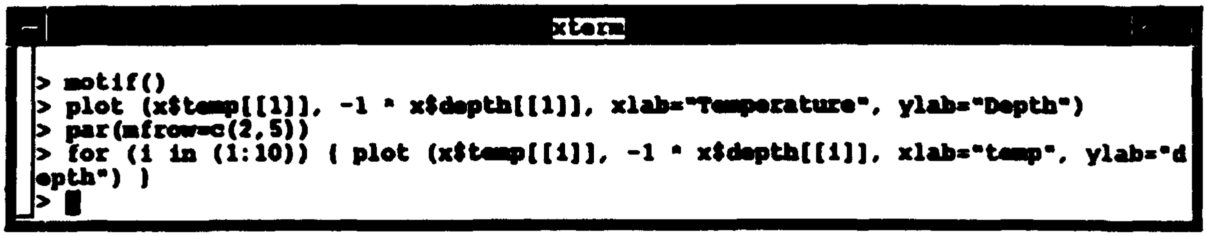


FIGURE 19. S-PLUS Plot (One Vector)

Figure 20 and Figure 21 plot the first 10 vectors.



```
> motif()
> plot (x$temp[[1]], -1 * x$depth[[1]], xlab="Temperature", ylab="Depth")
> par(mfrow=c(2,5))
> for (i in (1:10)) { plot (x$temp[[i]], -1 * x$depth[[i]], xlab="temp", ylab="d
epth") }
> 
```

FIGURE 20. Plotting Multiple Results

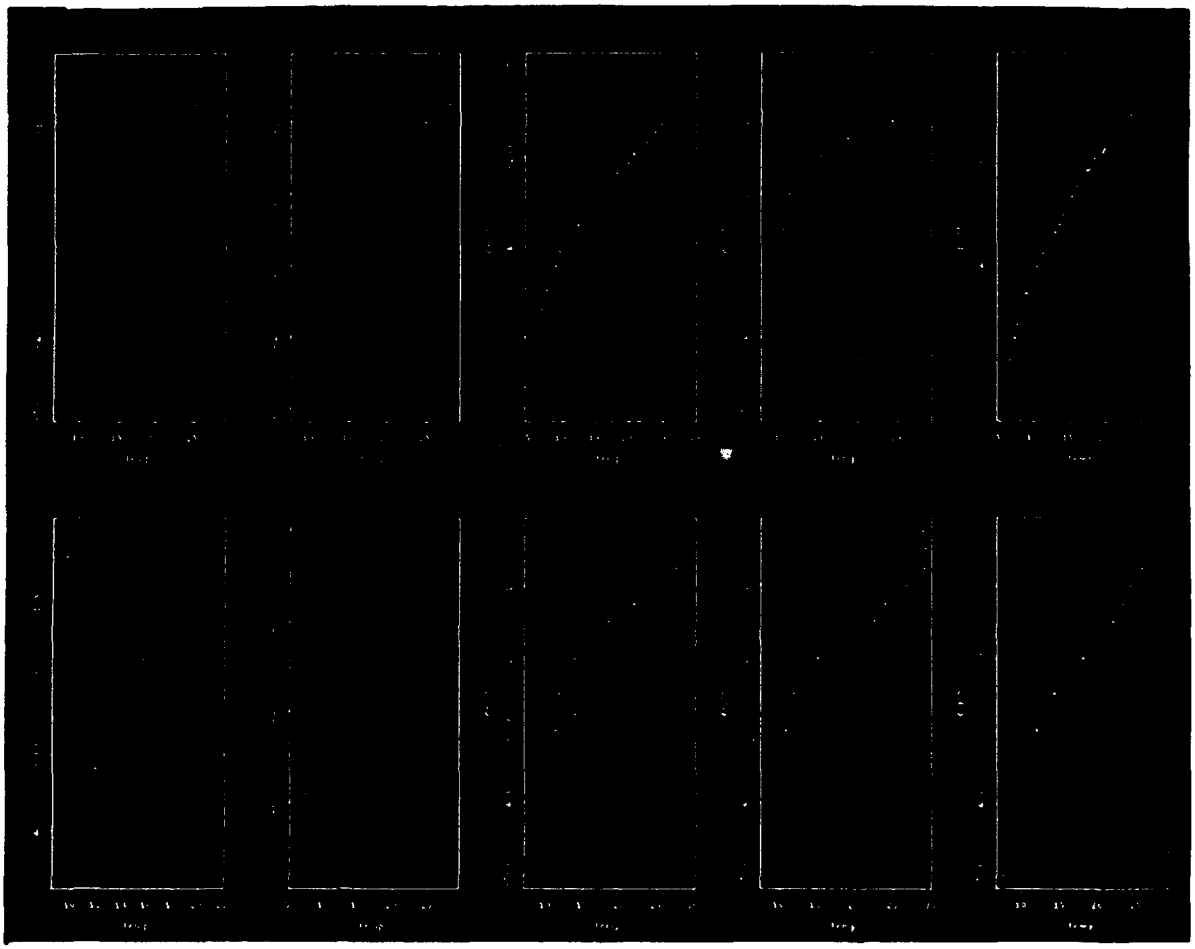
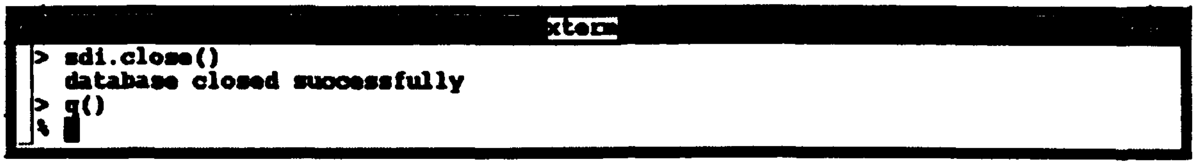


FIGURE 21. S-PLUS Plot (Ten Vectors)

10.5 Exiting S-PLUS

sdi.close() disconnects the S-PLUS session from the database. The commands in Figure 22 disconnect from the database and exit S-PLUS.



```
> sdi.close()
  database closed successfully
> q()
% █
```

FIGURE 22. Exiting S-PLUS

10.6 Transaction Management

Transaction management is implemented slightly differently in all the databases the S-PLUS database interface supports. The most notable difference is between Oracle and the other three databases (Montage, Postgres, and Sybase).

The first Oracle statement implicitly starts a transaction, which is not ended until a *commit* or *rollback* is executed. If queries executed by *sdi.submit()* change the database, those changes do not become permanent until a *commit* occurs. A *commit* makes all changes permanent as does any DDL statement such as create or drop. A *rollback* undoes all changes. *sdi.close()* commits all pending changes.

A transaction in Montage, Postgres, and Sybase must be explicitly started using the conventions of those databases.

11. FORTRAN Interface

The GDI FORTRAN interface provides database access from FORTRAN 77 applications. To use it, the user must know:

- The query language of the target database.
- The FORTRAN 77 Language.
- How to use the GDI functions and subroutines described in this section.

The software components listed below are referenced throughout this section. Contact your local system or database administrator to determine the actual location on your system:

libraries The main GDI library is named *libgdi.a*. Each database has its own additional library, named *libgdipg.a* for POSTGRES, *libgdiora.a* for ORACLE, and *libgdisyb.a* for SYBASE. Each database also has its own link file, named *pg_link.o* for POSTGRES, *ora_link.o* for ORACLE, and *syb_link.o* for SYBASE.

include files The GDI FORTRAN include file is named *gdi_f77.h* and must be included in all FORTRAN source code that executes GDI calls. It establishes a labelled common that contains standard codes for data types and error handling.

sample code Sample code is available in the GDI source code tree. For its exact location, contact your local system or database administrator. The Makefiles in this directory will be configured correctly for your installation.

11.1 Document Organization

This section is organized as follows:

Section 11.2	Summary of all GDI functions and subroutines
Section 11.3	Database connection
Section 11.4	Query execution
Section 11.5	Error handling
Section 11.6	Complete sample program
Section 11.7	Problem tracking
Section 11.8	Known problems and restrictions

11.2 Subroutine and Function Calls

This section summarizes the FORTRAN function and subroutine calls, sorted alphabetically by name.

The data type of each argument is listed in the right hand column. Character variables are of an arbitrary length.

Table 14. FORTRAN Data Types and Functions

Name	*Description*	*Type*
HEADER Variables	These header variables are defined in *gdi_f77.h*.	
	GDI DATA TYPES:	
	GDI_INT2	*integer*
	GDI_INT4	*integer*
	GDI_REAL4	*integer*
	GDI_REAL8	*integer*
	GDI_CHAR	*integer*
	GDI_STRING	*integer*
	GDI_UNDEFINED	*integer*
	ERROR HANDLING & DEBUGGING:	
	GDI_SUCCESS	*integer*
	GDI_FAILURE	*integer*
	GDI_NOMAP	*integer*
	GDI_NOCONN	*integer*
	GDI_DEBUG_OFF	*integer*
	GDI_DEBUG_ON	*integer*
	GDI_DEBUG_VERBOSE	*integer*
GDI_ADD_MAP_FIELD	INTEGER FUNCTION GDI_ADD_MAP_FIELD (DBCONN, MAP_ID, DB_NAME, PGM_NAME, DATA_TYPE, STR_LEN, ARRAY_LEN)	
	PURPOSE: Execute a database query.	
	INPUT ARGUMENTS:	
	DBCONN Database connect ID (see GDI_OPEN).	*integer*
	MAP_ID Query map ID (see GDI_OPEN_MAP).	*integer*
	DB_NAME Name of the database column in the retrieve/select list.	*char*
	PGM_NAME Name of the FORTRAN variable.	*char*
	DATA_TYPE GDI data type of PGM_NAME.	*integer*
	STR_LEN The length if DATA_TYPE is a GDI_STRING.	*integer*
	ARRAY_LEN If DATA_TYPE is an array, the number of elements in the array. This will always be 0 for ORACLE and SYBASE.	*integer*
	RETURN: GDI_SUCCESS or GDI_FAILURE.	*integer*

Table 14. FORTRAN Data Types and Functions

Name	*Description*	*Type*
GDI_CLOSE	INTEGER FUNCTION GDI_CLOSE (DBCONN) *PURPOSE:* Close the specified database connection. *INPUT ARGUMENTS:* DBCONN Database connect ID (see GDI_OPEN). *RETURN:* GDI_SUCCESS or GDI_FAILURE.	 *integer* *integer*
GDI_CLOSE_MAP	SUBROUTINE GDI_CLOSE_MAP (DBCONN, MAP_ID) *PURPOSE:* Ends definition for a query mapping. *INPUT ARGUMENTS:* DBCONN Database connect ID (see GDI_OPEN). MAP_ID Query map ID (see GDI_OPEN_MAP).	 *integer* *integer*
GDI_DESTROY_MAP	SUBROUTINE GDI_DESTROY_MAP (DBCONN, MAP_ID) *PURPOSE:* Destroys mapping. *INPUT ARGUMENTS:* DBCONN Database connect ID (see GDI_OPEN). MAP_ID Query map ID (see GDI_OPEN_MAP).	 *integer* *integer*
GDI_ERROR_GET	SUBROUTINE GDI_ERROR_GET (DBCONN, ERRCODE, ERRTEXT, MAXTEXT, STATUS, SEVERITY) *PURPOSE:* Retrieve the error from the GDI error handler. *INPUT ARGUMENTS:* DBCONN Database connect ID (see GDI_OPEN). MAXTEXT Length of ERRTEXT variable. Database message text longer than this will be truncated. *OUTPUT ARGUMENTS:* ERRCODE Error code. ERRTEXT Error message. STATUS GDI error status (GDI_SUCCESS or GDI_FAILURE). SEVERITY GDI severity level (GDI_NOERROR, GDI_WARNING, or GDI_FATAL).	 *integer* *integer* *integer* *char* *integer* *integer*

Table 14. FORTRAN Data Types and Functions

Name	*Description*	*Type*
GDI_ERROR_INIT	SUBROUTINE GDI_ERROR_INIT (DBCONN, DEBUG, THRESHOLD, RESERVED1, RESERVED2)	
	PURPOSE: Initialize error handling flags.	
	INPUT ARGUMENTS:	
	DBCONN Database connect ID (see GDI_OPEN).	*integer*
	DEBUG Default setting is GDI_DEBUG_OFF. GDI_DEBUG_ON causes error messages to be output to *stderr*. GDI_DEBUG_VERBOSE may cause additional messages to be output.	*integer*
	THRESHOLD Controls how severe an error must be in order to cause failure. The default setting is GDI_WARNING, which means that warning and fatal errors both return GDI_FAILURE to the calling routine. If set to GDI_FATAL, then only fatal errors return GDI_FAILURE; warnings return GDI_SUCCESS.	*integer*
	RESERVED1 Currently not used.	*integer*
	RESERVED2 Currently not used.	*integer*
GDI_INIT	INTEGER FUNCTION GDI_INIT (APPNAME)	
	PURPOSE: Initialize the GDI.	
	INPUT ARGUMENTS:	
	APPNAME: Program name.	*char*
	RETURN: GDI_SUCCESS or GDI_FAILURE	*integer*
GDI_OPEN	INTEGER FUNCTION GDI_OPEN (VENDOR, ACCOUNT, PASSWORD, DATABASE, SERVER, APPNAME)	
	PURPOSE: Open a connection to a database.	
	INPUT ARGUMENTS:	
	VENDOR Database vendor name; currently includes *oracle* or *postgres*.	*char*
	ACCOUNT Database account or user name.	*char*
	PASSWORD Password for the account.	*char*
	DATABASE Database name.	*char*
	SERVER Server name (Sybase & Postgres only).	*char*
	APPNAME Program name.	*char*
	RETURN: Database connection ID. GDI_NOCONN means it failed.	*integer*

Table 14. FORTRAN Data Types and Functions

Name	*Description*	*Type*
GDI_OPEN_MAP	INTEGER FUNCTION GDI_OPEN_MAP (DBCONN)	
	PURPOSE: Establishes the relationship between database query columns and FORTRAN variables.	
	INPUT ARGUMENTS:	
	DBCONN Database connect ID (see GDI_OPEN).	*integer*
	RETURN: Query map id. GDI_NOMAP means it failed.	*integer*
GDI_SUBMIT	INTEGER FUNCTION GDI_SUBMIT (DBCONN, MAP_ID, QUERY, MAXRECS, RETRIEVED, AFFECTED, MORE_DATA)	
	PURPOSE: Execute a database query.	
	INPUT ARGUMENTS:	
	DBCONN Database connect ID (see GDI_OPEN).	*integer*
	MAP_ID Query map ID (see GDI_OPEN_MAP).	*integer*
	QUERY Character string containing a complete database query.	*char*
	MAXRECS Controls how many instances are retrieved. Should be set to the maximum number of records that can fit into the FORTRAN variable.	*integer*
	OUTPUT ARGUMENTS:	
	RETRIEVED Records the number of records retrieved.	*integer*
	AFFECTED Records the number of records affected by the query.	*integer*
	MORE_DATA If the data available is greater than MAXRECS, MORE_DATA will be set to TRUE.	*logical*4*
	RETURN: GDI_SUCCESS or GDI_FAILURE.	*integer*
GDI_TRACE	SUBROUTINE GDI_TRACE (DBCONN, STATE, FILENAME)	
	PURPOSE: Turns database-specific debug on/off.	
	INPUT ARGUMENTS:	
	DBCONN Database connect ID (see GDI_OPEN).	*integer*
	STATE TRUE turns trace on, FALSE turns it off.	*integer*
	FILENAME Output filename (SYBASE only).	*char*

11.3 Connecting to a Database

This section describes how to initialize the GDI with *GDI_INIT()*, connect to a database with *GDI_OPEN()* and disconnect from a database with *GDI_CLOSE()*.

GDI_INIT() initializes the GDI to communicate with the database(s) to which a program will connect. *GDI_OPEN()* establishes a connection to the database. *GDI_OPEN()* arguments were described in detail in Section 11.2. But since not all databases use all arguments, Table 15 summarizes which databases use each parameter.

Table 15. *GDI_OPEN()* Parameters

Parameter	*ORACLE*	*POSTGRES*	*SYBASE*
vendor	yes	yes	yes
account	yes	no	yes
password	optional	no	yes
database	optional	optional	yes
server	no	optional	yes
appname	no	no	yes

Some *GDI_OPEN()* parameters are optional.

For ORACLE, *password* is not applicable to ops$ logins (logins tied to operating system accounts). Also the entire account/password connect string may be sent in *via* the *account* parameter.

For POSTGRES, if *database* is not set, the connection will be set from the PGDATABASE environmental variable. If *server* is not set, it will be set from the PGHOST environmental variable.

GDI_OPEN() returns an integer database connection handle that is used by other GDI calls; its main purpose is to store error information. If it is equal to GDI_NOCONN, it means that the connection failed. Example 17 initializes the GDI and establishes a connection to a POSTGRES database.

Example 17:

```
C        === Initialize the GDI and connect to POSTGRES database 'demo' ===

         include '../../include/gdi_f77.h
         character*30     VENDOR, DBNAME, DBHOST, na
         integer          DBCONN, STATUS

C        === Initialize program variables ===

         VENDOR = 'postgres'
         DBNAME = 'demo'
         DBHOST = 'heel.s2k.berkeley.edu'
         NA = ' '
```

```
C         === Initialize GDI ===

          STATUS = GDI_INIT ('sample')

C         === OPEN DATABASE CONNECTION ===

          DBCONN = GDI_OPEN (VENDOR, NA, NA, DBNAME, DBHOST, NA)
          IF (DBCONN .EQ. GDI_NOCONN) THEN
               ... handle error, described in Section 11.5...
          END IF
```

If the *database* and *server* parameters are set in the PGDATABASE and PGHOST environmental variables, all parameters to *GDI_OPEN()*, except for *vendor*, can be blank.

GDI_CLOSE() disconnects an application from the database, demonstrated in Example 18.

Example 18:

```
C         === Disconnect from the database ===

          STATUS = GDI_CLOSE (DBCONN)
```

11.4 Executing Queries

GDI_SUBMIT() executes a database query and returns GDI_SUCCESS if the query succeeded and GDI_FAILURE if it did not.

The GDI distinguishes between queries that return data, as with a POSTQUEL retrieve or a SQL select, and queries that do not return data. First we will look at queries that do not return data results.

11.4.1 Queries that Do Not Return Data

Example 19 creates two classes in a POSTGRES database.[1]

Example 19:

```
          character*100     QUERY
C         This is not a retrieve so set MAP_ID and MAXRECS to 0.
          integer           MAP_ID=0, MAXRECS=0
          integer           ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA

C     ========== CREATE cnsierra CLASS ==========

          QUERY=    'create cnsierra (year=int4, julday=int4, precip=int4,' //
     &              'tmax=float4, 'tmin=float4, tmean=float4)'
          STATUS=GDI_SUBMIT (DBCONN, MAP_ID, QUERY, MAX_RECS,
     &                      ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA)

C         ========== CREATE sst CLASS ==========

          QUERY=    'create sst (lat=float4, long=float4, time=float8,' //
     &              'temp=float4[6]'
          STATUS=GDI_SUBMIT (DBCONN, MAP_ID, QUERY, MAX_RECS,
     &                      ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA)
```

GDI_SUBMIT() executes any query. Example 20 loads data into *cnsierra*, then updates one of its attributes.

Example 20:

```
          QUERY=    'copy cnsierra from /usr/data/cnsierra.dat'
          STATUS=GDI_SUBMIT (DBCONN, MAP_ID, QUERY, MAX_RECS)

          QUERY=    'replace cnsierra (cnsierra.precip= -9.99) ' //
     &              'where cnsierra.precip=0'

          STATUS=GDI_SUBMIT (DBCONN, MAP_ID, QUERY, MAX_RECS,
     &                      ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA)
```

After an update, ROWS_AFFECTED should report the number of rows that were updated. Currently this does not work for POSTGRES databases.

[1] Example queries are from the *Introductory Guide to POSTGRES* by Emelia C. Villaros-Bainto.

11.4.2 Queries That Return Data

A query that returns data from the database has two steps:

1. Map each column in the query's retrieve list to a FORTRAN variable.
2. Execute the query with *GDI_SUBMIT()*.

GDI_CREATE_MAP(), demonstrated in Example 21, allocates a mapping to establish relationships between a query column and FORTRAN variables. It returns a MAP_ID, which is used in the other mapping calls.

Example 21:

```
C         ============ Create a query mapping =============

          INTEGER MAP_ID

          MAP_ID = GDI_OPEN_MAP (DBCONN)
          IF (MAP_ID .EQ. GDI_NOMAP) THEN
               WRITE (6,*) 'GDI_OPEN_MAP failed.'
          END IF
```

GDI_ADD_MAP_FIELD(), demonstrated in Example 22, matches a database result column to a FORTRAN variable. Each column in a query must have a corresponding mapped FORTRAN variable.

Example 22:

```
C         =========== Map Database Columns to FORTRAN variables ====

          REAL       LATITUDE(100), TEMP(6,100)
          REAL*8     TIME(100)
          CHAR*80    QUERY

          QUERY = 'retrieve s.latitude, s.temp, s.time) from s in sst'

          STATUS = GDI_ADD_MAP_FIELD (DBCONN, MAP_ID, 'latitude',
     &         LATITUDE, GDI_REAL4, 0, 0)

          STATUS = GDI_ADD_MAP_FIELD (DBCONN, MAP_ID, 'temp',
     &         TEMP, GDI_REAL4, 0, 6)

          STATUS = GDI_ADD_MAP_FIELD (DBCONN, MAP_ID, 'time',
     &         TIME, GDI_REAL8, 0, 0)
```

Note that the *temp* attribute in Example 22 is a POSTGRES array attribute containing 6 values. This syntax is only valid for POSTGRES databases. Currently array support is limited to 2 dimensional arrays, and variables must be declared carefully. The size of the POSTGRES array must be the first dimension, as in *TEMP(6, 100)*. The number of rows is the second dimension.

GDI_CLOSE_MAP(), demonstrated in Example 23, ends the definition for a mapping.

Example 23:

```
C         ========= End Query Mapping ================

          CALL GDI_CLOSE_MAP (MAP_ID)
```

GDI_DESTROY_MAP(), demonstrated in Example 24, drops the mapping relationship, freeing all local memory allocated.

Example 24:

```
C        ============== Drop Query Map ===================

         CALL GDI_DESTROY_MAP (DBCONN, MAP_ID)
```

The MAP_ID does not have to be destroyed after executing a query. It may be reused in subsequent queries so long as the number of columns do not change or the data types of the columns do not change.

Once the mapping has been established, the query may be executed with *GDI_SUBMIT()*, demonstrated in Example 25.

Example 25:

```
C        ============= Execute the Query =================

         integer        MAXRECS, ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA

         MAXRECS = 100

         STATUS=GDI_SUBMIT (DBCONN, MAP_ID, QUERY, MAXRECS,
     &                  ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA)
```

MAXRECS indicates the maximum number of instances or rows of data that should be returned. It must not be set higher than the array lengths of the FORTRAN variables. The number of rows actually retrieved will be stored in ROWS_RETRIEVED. If more data are available than MAXRECS, the MORL _DATA flag will be set to TRUE.

11.5 Handling Errors

Some GDI functions, such as *GDI_OPEN()* and *GDI_OPEN_MAP()* return an integer handle that should be greater than 0 if the call succeeded. All other GDI functions return GDI_SUCCESS or GDI_FAILURE.

GDI_ERROR_GET() retrieves specific error information. Example 26 calls *GDI_ERROR_GET()* after detecting an error.

Example 26:

```
                character*80     ERRTXT
                integer          DBCONN, DBERR, SEVERITY

                DBCONN = GDI_OPEN (VENDOR, na, na, DBNAME, na, na)
                IF (DBCONN .EQ. GDI_NOCONN) THEN
                     CALL GDI_ERROR_GET (DBCONN, DBERR, ERRTXT, 80, STATUS,
       &                  SEVERITY)
                     WRITE(0, *) ERRTXT
                     ....... handle error .....
                END IF
```

GDI_ERROR_INIT() initializes two error handling flags, *debug* and *threshold*. *debug* and *threshold* may be changed at any time. Example 27 sets *debug* to GDI_DEBUG_VERBOSE and *threshold* to GDI_WARNING.

Example 27:

```
  c             === Output verbose debug messages & treat warnings as fatal ===

                CALL GDI_ERROR_INIT (DBCONN, GDI_DEBUG_VERBOSE, GDI_WARNING)
```

GDI_TRACE() turns database vendor-specific tracing on and off and may be called at any time. Example 28 turns trace on.

Example 28:

```
  c             === Turn database tracing on ===

                CALL GDI_TRACE (DBCONN, TRUE, FILENAME)
```

11.6 Sample Programs

This section includes complete sample FORTRAN programs. Example 29 is a POSTGRES example.

Example 29:

```
C         ========== Sample POSTGRES program ================

include '../../include/gdi_f77.h'

C         define local variables

C         ========== Connect to database ======================

          CHARACTER*10        VENDOR, DATABASE, NA
          CHARACTER*16        PRGNAM
          INTEGER             DBCONN

C         ========== Error handling variables ===================

          CHARACTER*80        ERRTXT
          INTEGER             MAXTXT, STATUS, SEVERITY, ERRCDE

C         ========== Query variables ==========================

          INTEGER*4           MAP_ID
          CHARACTER*80        QUERY
          INTEGER             MAXRECS, ROWS_RETRIEVED, ROWS_AFFECTED
          INTEGER             ROWS_LEFT
          LOGICAL             MORE_DATA

C         ============ Output Variables ==========================

          REAL*8              TIME(20)
          INTEGER             NSAMP(20)
          CHARACTER*16        STA(20)
          INTEGER             I

          VENDOR = 'postgres'
          DATABSE = 'geodemo'
          PRGNAM = 'gdi_f77_pg_test'
          MAXRECS = 20
          MAXTXT = 80

C         Some GDI_OPEN arguments are Not Applicable (NA) to POSTGRES

          NA = ' '

C         ================== Initialize the GDI.=====================

          STATUS = GDI_INIT (PRGNAM)
          IF (STATUS .NE. GDI_SUCCESS) THEN
                WRITE (6,*) 'GDI_INIT Failed. Program exiting.'
                GOTO 999
          END IF
```

```
C          =============== Open a connection to the database.==========

           DBCONN = GDI_OPEN (VENDOR, NA, NA, DATABASE, NA, PRGNAM)
           IF (DBCONN .EQ. GDI_NOCONN) THEN
                CALL GDI_ERROR_GET (DBCONN, ERRCDE, ERRTXT, MAXTXT,
     &               STATUS, SEVERITY)
                WRITE (6,*) 'GDI_OPEN Failed: Error Code ', ERRCDE
                WRITE (6,*) ERRTXT
                GOTO 999
           END IF

C          Setting GDI_DEBUG_ON prints errors to the screen.

           CALL GDI_ERROR_INIT (DBCONN, GDI_DEBUG_ON,GDI_WARNING,
     &          RESERVED1, RESERVED2)

C          ================ Build a query.==============================

           QUERY = 'retrieve (w.time, w.nsamp, w.sta) from w in wfdisc'

C          ================ Create query mapping.====================

           MAP_ID = GDI_OPEN_MAP (DBCONN)
           IF (MAP_ID .EQ. GDI_NOMAP) THEN
                GOTO 999
           END IF

C          === Map each attribute being retrieved to a FORTRAN variable. ===

           STATUS = GDI_ADD_MAP_FIELD (DBCONN, MAP_ID,
     &          'time', TIME, GDI_REAL8, 0, 0)
           IF (STATUS .NE. GDI_SUCCESS) THEN
                GOTO 999
           END IF

           STATUS = GDI_ADD_MAP_FIELD (DBCONN, MAP_ID,
     &          'nsamp', NSAMP, GDI_INT4, 0, 0)
           IF (STATUS .NE. GDI_SUCCESS) THEN
                GOTO 999
           END IF

           STATUS = GDI_ADD_MAP_FIELD (DBCONN, MAP_ID,
     &          'sta', STA, GDI_STRING, 16, 0)
           IF (STATUS .NE. GDI_SUCCESS) THEN
                GOTO 999
           END IF

           CALL GDI_CLOSE_MAP(DBCONN, MAP_ID)

C          =============== Execute the query =========================

           STATUS = GDI_SUBMIT(DBCONN, MAP_ID, QUERY, MAXRECS,
     &          ROWS_RETRIEVED, ROWS_AFFECTED, MORE_DATA)
           IF (STATUS .NE. GDI_SUCCESS) THEN
                GOTO 999
           END IF
```

```
C           ================ Print out retrieved data. ==================

            WRITE (6,*) ROWS_AFFECTED, ' rows satisfied the query.'
            WRITE (6,*) ROWS_RETRIEVED, ' rows were retrieved.'
            DO 10 I = 1, ROWS_RETRIEVED
                WRITE (6,*) STA(I), TIME(I), NSAMP(I)
10          CONTINUE

            IF (MORE_DATA) THEN
                ROWS_LEFT = ROWS_AFFECTED - ROWS_RETRIEVED
                WRITE (6,*) ROWS_LEFT, ' more rows are available.'
            ELSE
                WRITE (6,*) 'No more data exists in the database.'
            END IF

C           ================ Destroy query mapping. ==================

            CALL GDI_DESTROY_MAP (DBCONN, MAP_ID)

999         STATUS = GDI_CLOSE (DBCONN)
            END
```

When run on a database containing seismic data, output looks like this:

```
% gdi_f77_pg_test

      63 rows satisfied the query.
      20 rows were successfully retrieved from the database.

BLA            636710425.00000      14280
MOX            636710786.05000      1180
MO             636710786.05000      1180
MO             636710786.05000      1180
WRA            636710849.49200      2400
WRA            636710849.49200      2400
ASAR           636710887.89900      2400
ASAR           636710887.89900      2400
ARA0           636711023.70900      4797
ARA0           636711023.70900      4800
LTX            636711827.00000      10320
GRF            636713180.00000      2400
GRF            636713559.00000      2400
KBA            636713564.00200      12000
ASAR           636713609.66400      2400
ASAR           636713609.66400      2400
NRA0           636713630.60300      4792
NRA0           636713630.60300      4800
GRF            636713920.00000      2400
GAR            636713921.69900      2400

      43 more rows are available.
```

11.7 Troubleshooting Tips

Here are a few tips for when things do not work as expected:

- Test database queries interactively before putting them into a program.
- GDI_ERROR_INIT with the debug flag set to GDI_DEBUG_ON outputs errors to the screen.
- GDI_ERROR_INIT with the debug flag set to GDI_DEBUG_VERBOSE outputs debug messages to the screen.
- GDI_TRACE set to TRUE outputs database-specific debugging messages.

11.8 Current Restrictions

POSTGRES

- *GDI_SUBMIT()*
  ROWS_AFFECTED will not be set unless the command was an APPEND.

  Bulit-in Types
  The following built-in types are not directly supported yet. The GDI will return these types as strings to the application.

  large objects

  types composed of a structure, such as box and polygon

- *User-Defined Types*
  The following SEQUIOA types are handled:

  char2

  char4

  char8

  Adding new types requires changing source code and recompiling. We are working on a strategy to dynamically manage types.

- *Database Nulls*
  If a database attribute is NULL (*i.e.*, it does not have a value), the output variable will be assigned a value as follows:

GDI_INT2, GDI_INT4:	0
GDI_REAL4, GDI_REAL8:	0.0
GDI_STRING:	blank padded to the size of the FORTRAN variable
GDI_CHAR:	blank

- *Named Columns*
  The GDI cannot determine the type of some named columns.

Instead of this:	retrieve (my_name=p.name) from p in foo
Do this:	retrieve(p.name) from p in foo

Part IV: Reference Manual

NAME

gdi_gen_Astructs – tool to generate header files containing structure
declarations for the GDI's ArrayStructs constructor.

SYNOPSIS

gdi_gen_Astructs par=gdi_gen_Astructs.par

PAR PARAMETERS

account database account/password and connect string if required

vendor database vendor name

query syntactically correct sql statement, NO where clause

structname name of the structure to be generated, first letter capitalized by convention

DESCRIPTION

This tool creates data structures based on the columns resulting from a database query and outputs them to a header file. The structures usually correspond to a table structure but could be a sub or superset of any combination of relations. Queries are submitted with gdi_submit(). The ArrayStructs constructor and the header generated by **gdi_gen_Astructs** emulate libdb30 style array fetches in that the tuples are returned in an array of structures. See gdi_submit() for a complete description of how to fetch data with the GDI.

One of the data structures contains "NA" values for each attribute or column. These values are obtained from the database table *na_value*. The na_value table has 2 fields, attribute and na_value. Both are of type char(30). The not available value for a specific attribute can be stored in this table. If the attribute does not exists in *na_value* or the table does not exist, default values are used. The default for ints and floats are -1 and -999.0. The default for a string is a "-".

The select list of queries using the generated header file must correspond to that of the query used to create the structures. Every column in the query must have a column of the same name and type in the header file. The columns in the select list may be a subset of the original list and may appear in any order.

The header files may be used in conjunction with gdi_add_ArrayStructs() and gdi_get_ArrayStructs(). These functions provide a layer around gdi_submit(), gdi_insert(), and the dbObj. gdi_get_ArrayStructs() submits the query and retrieves the array of tuples from the dbObj. The dbObj is freed by the function and the array of tuples is returned to the calling application. It is the responsibility of the application to free the results. gdi_add_ArrayStructs() takes an array of tuples and inserts them into a database table. The dbObj required by gdi_insert() is created by the function and destroyed before the function returns. See **tst_ArrayStructs_submit** and **tst_ArrayStructs_insert** in libgendb/test for usage.

The sample *parfile* below would generate arrival_Astructs.h:

```
account="realtime/realtime@t:troll:dev6033"
vendor="oracle"
query="SELECT * from arrival"
structname="Arrival"
```

DIAGNOSTICS

GDI_SUCCESS

No problem generating the header file.

GDI_FAILURE

An error occurred.

FILE

gdi_gen_ArrayStructs.c

NOTES

Not implemented for FORTRAN.

SEE ALSO

gdi_insert(3), **gdi_submit**(3), **gdi_add_ArrayStructs**(3), **gdi_get_ArrayStructs**(3), libdb30: **array_fetch**(3)

AUTHOR

Mari Mortell, SAIC Geophysical Systems Operation November 1991

NAME

gdi_abort – abort the current command

SYNOPSIS

```
#include "libgdi.h"

int
gdi_abort (conn)
dbConn          *conn;          /* (i) database connection */
```

DESCRIPTION

gdi_abort() cancels all query activity on a given *dbConn*; however, behavior may be vendor dependent. For ORACLE, if no command is currently executing and the next routine is a fetch, the fetch will be asynchronously aborted. For SYBASE and MONTAGE, commands on all query channels associated with the dbConn will be cancelled. gdi_aobrt() has no effect for POSTGRES.

ARGUMENTS

conn The database connector for the connection which the channel was opened on.

DIAGNOSTICS

gdi_abort() returns one of the following status values:

GDI_SUCCESS
Abort succeeded.

GDI_FAILURE
Abort failed; possibly the database connection dropped.

FILE

gdi_abort.c

SEE ALSO

gdi_flush(3)

AUTHOR

Jean T. Anderson, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_add_ArrayStructs – Insert an array of structures into a database table.

SYNOPSIS

```
#include "libgdi.h"
#include "<type>_Astructs.h"

int
gdi_add_ArrayStructs (conn, table_name, array, ntuple, type)
dbConn            *conn;          /* (i) database connection */
char              *table_name;    /* (i) database table */
void              *array;         /* (i) array of structs */
int               ntuple;         /* (i) number of tuples in the array */
ArrayStructsArgs  *type;          /* (i) structure definition */
```

DESCRIPTION

gdi_add_ArrayStructs() inserts the data in an array of structures into a database table. Headers containing a structure definition with fields corresponding to the columns of the table are created with gdi_gen_Astructs(1). Although the structure may only contain fields that correspond to columns in the database table, the order of the fields in the structure need not match the order of the columns in the table.

ARGUMENTS

conn The database connector.

table_name The database table into which the data is to be inserted.

array The array of structures containing the data to be inserted into the database.

ntuple The number of tuples in the array.

type A description of the array structure, the "NA" values and other information needed to process the array for input. The description is contained in the "<type>_Astructs.h" header.

EXAMPLE

The following example uses a header dumped by gdi_gen_Astructs(1) using the query, "select * from arrival". The structure definition in arrival_Astructs.h is shown below.

```
typedef struct arrival {
            char        sta [7];
            double      time;
            long        arid;
            long        jdate;
            long        stassid;
            long        chanid;
            char        chan [9];
            char        iphase [9];
            char        stype [2];
            double      deltim;
            double      azimuth;
            double      delaz;
            double      slow;
            double      delslo;
            double      ema;
            double      rect;
            double      amp;
```

```
                double      per;
                double      logat;
                char        clip [2];
                char        fm [3];
                double      snr;
                char        qual [2];
                char        auth [16];
                long        commid;
                char        lddate [18];
        } Arrival;
```

The following code segment inserts data into the database.

```
        #include "libgdi.h"
        #include "arrival_Astructs.h"

        ...

        dbConn      *conn;                          /* database connector */
        char        *table = "arrival";
        Arrival     *tuples;                        /* array of tuples */
        int         ntuples = 10;                   /* number of tuples in the array */

        int         err_code;                       /* error handling variables */
        char        err_text [200];
        dbStatus    status;
        dbErrLev    severity;

        ... initialize the GDI, open a database connection ...

        ... create an array of tuples ...

        if ((ntuples = gdi_add_ArrayStructs (conn, table, (void *) tuples, ntuples,
                                    &ARRIVAL_CONTAINER_DEF)) < 0)
        {
                    gdi_error_get (conn, &err_code, err_text, sizeof (errtext),
                                    &status, &severity);

                    ... handle the error ...
        }
```

DIAGNOSTICS

gdi_add_ArrayStructs() returns the number of tuples inserted if successful, otherwise it returns -1. Error codes and messages may be retrieved from the database connector with gdi_error_get(3).

FILE

gdi_ArrayStructs.c, gdi_ArrayStructs.h

SEE ALSO

gdi_error_get(3), gdi_gen_Astructs(1), gdi_get_ArrayStructs(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_auto_commit – Enable or disable auto commit mode

SYNOPSIS

```
#include "libgdi.h"

int
gdi_auto_commit (conn, mode)
dbConn          *conn;          /* (i) database connection */
int             mode;           /* (i) auto commit mode, TRUE or FALSE */
```

DESCRIPTION

A database transaction is a statement, or statements, treated as an atomic unit. If auto commit is enabled, each database statement is treated as a transaction and the results are automatically committed when the statement is executed. The auto commit mode is controlled at the connector level (rather than the channel level).

Note that the ability to enable or disable the auto commit mode is only implemented for ORACLE connections. The auto commit default mode for ORACLE connections is OFF. SYBASE always commits the results of each statement at execution time (essentially auto commit is ON) unless gdi_begin_tran(3) has been called.

The state of the auto commit mode for a connection may be ascertained through the GDI_AUTOCOM_ON(conn) macro.

ARGUMENTS

conn The database connector.

mode The auto commit mode to be set. TRUE enables auto commit. FALSE disables auto commit.

DIAGNOSTICS

gdi_auto_commit() returns one of the following status values:

GDI_SUCCESS
Operation succeeded.

GDI_FAILURE
Operation failed; possibly the connection dropped.

FILE

gdi_tran.c

SEE ALSO

gdi_begin_tran(3), **gdi_commit**(3), **gdi_rollback**(3), **gdi_savepoint**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_begin_tran – Explicitly begin a transaction

SYNOPSIS

```
#include "libgdi.h"

int
gdi_begin_tran (conn, channo, tran_name)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
char            *tran_name;     /* (i) transaction name */
```

DESCRIPTION

A database transaction is a statement, or statements, treated as an atomic unit. gdi_begin_tran() explicitly begins a transaction. The transaction is ended by a gdi_commit() or gdi_rollback(). A transaction acquires *locks* on data as it queries or updates the database. The locks acquired during a transaction are released at the next commit or rollback. Transactions should be as tight and small as possible so lock resources needed by other database processes are released back to the system.

Transaction management is implemented slightly differently in all the databases the gdi supports. gdi_begin_tran() currently has no affect on ORACLE databases since the first ORACLE statement implicitly starts a transaction, which is not ended until a gdi_commit() or gdi_rollback() occurs.

ARGUMENTS

conn The database connector.

channo The channel number (SYBASE and MONTAGE). SYBASE transactions are handled at the DBPROCESS level. MONTAGE transactions are handled at the database connection level, but each gdi query channel maps to a separate database connection. The channel argument is ignored for ORACLE and POSTGRES.

tran_name Transaction name of the transaction to be started. This argument is only valid for SYBASE which allows nested, named transactions.

DIAGNOSTICS

gdi_begin_tran() returns one of the following status values:

GDI_SUCCESS

Operation succeeded.

GDI_FAILURE

Operation failed; possibly the connection dropped.

FILE

gdi_tran.c

NOTES

Not implemented in INGRES yet.

SEE ALSO

gdi_commit(3), gdi_get_dboption(3), gdi_rollback(3), gdi_savepoint(3), gdi_set_dboption(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_channel_is_open – is channel open?

SYNOPSIS

```
#include "libgdi.h"

int
gdi_open_channel (conn, channo)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
```

DESCRIPTION

gdi_channel_is_open() returns TRUE if a given channel is open, or FALSE if it is not.

ARGUMENTS

conn The database connector for the connection the channel was opened on.

channo Channel number of the channel to be checked.

DIAGNOSTICS

gdi_channel_is_open() returns one of the following status values:

TRUE Channel is open.

FALSE Channel is not open.

FILE

gdi_channel.c

SEE ALSO

gdi_close_channel(3), gdi_open_channel(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_close – close the specified database connection

SYNOPSIS

```
#include "libgdi.h"

int
gdi_close (conn)
dbConn          *conn;          /* (i) database connection */
```

DESCRIPTION

gdi_close() closes a specific connection to the database and frees the *dbConn* structure.

ARGUMENTS

conn The database connector for the connection to be closed.

DIAGNOSTICS

gdi_close() returns one of the following status values:

GDI_SUCCESS

Connection successfully closed.

GDI_FAILURE

Not connected to database.

FILE

gdi_conn.c

SEE ALSO

gdi_open(3), gdi_dead(3), gdi_exit(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_close_channel – close a database channel

SYNOPSIS

```
#include "libgdi.h"

int
gdi_close_channel (conn, channo)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
```

DESCRIPTION

gdi_close_channel() closes a specified channel.

ARGUMENTS

conn The database connector for the connection the channel was opened on.

channo Channel number of the channel to be closed.

DIAGNOSTICS

gdi_close_channel() returns one of the following status values:

GDI_SUCCESS
Succeeded in closing channel.

GDI_FAILURE
Could not close channel, possibly because the connection dropped.

FILE

gdi_channel.c

SEE ALSO

gdi_channel_is_open(3), gdi_open_channel(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_commit – commit current transaction

SYNOPSIS

```
#include "libgdi.h"

int
gdi_commit (conn, channo, tran_name)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
char            *tran_name;     /* (i) transaction name */
```

DESCRIPTION

A database transaction is a statement, or statements, treated as an atomic unit. gdi_commit() ends the current transaction by applying all changes to the database.

ARGUMENTS

conn The database connector.

channo The channel number (SYBASE and MONTAGE). SYBASE transactions are handled at the DBPROCESS level. MONTAGE transactions are handled at the database connection level, but each gdi query channel maps to a separate database connection. The channel argument is ignored for ORACLE and POSTGRES.

tran_name Transaction name of the transaction to be committed. This argument is only valid for SYBASE which allows nested, named transactions.

DIAGNOSTICS

gdi_commit() returns one of the following status values:

GDI_SUCCESS
Commit succeeded.

GDI_FAILURE
Commit failed; possibly the connection dropped.

FILE

gdi_tran.c

SEE ALSO

gdi_rollback(3), gdi_savepoint(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_dead – determines if a database connection is dead or live

SYNOPSIS

```
#include "libgdi.h"

int
gdi_dead (conn, channo)
dbConn          *conn;          /* (i) database connection */
int             *channo;        /* (i) database channel number */
```

DESCRIPTION

gdi_dead() pings the database to determine if a database connection is still established.

ARGUMENTS

conn The database connector for the connection to be tested.

channo The database channel number for the channel to be tested.

DIAGNOSTICS

gdi_dead() returns one of the following status values.

GDI_SUCCESS
Connection to database is OK.

GDI_FAILURE
Not connected to database.

SEE ALSO

gdi_close(3), **gdi_exit**(3), **gdi_open**(3)

AUTHOR

Jean T. Anderson, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_error_flags – retrieve debug and threshold settings

SYNOPSIS

```
#include "libgdi.h"

int
gdi_error_flags (conn, debug, threshold)
dbConn          *conn;          /* (i) database connector */
int             *debug;         /* (o) GDI_DEBUG_ON, GDI_DEBUG_OFF, or GDI_DEBUG_VERBOSE */
int             *threshold;     /* (o) GDI_WARNING or GDI_FATAL */
```

DESCRIPTION

Errors are handled on a connection by connection basis. gdi_error_flags() retrieves the current settings of *debug* and *threshold* for a specified connection.

ARGUMENTS

conn The database connector. If NULL, gets global error flags.

debug GDI_DEBUG_OFF by default, if set to GDI_DEBUG_ON, errors are output automatically to *stderr*. GDI_DEBUG_VERBOSE causes numerous debug messages as well as errors and warnings to be output to *stderr*.

threshold Controls the threshold at which an error or warning causes a GDI_FAILURE. A threshold of GDI_WARNING causes all warnings and errors to be interpreted as failures. A threshold of GDI_FATAL causes only fatal errors to be interpreted as failures.

DIAGNOSTICS

gdi_error_flags() always returns GDI_SUCCESS.

FILE

gdi_error.c

SEE ALSO

gdi_error_get(3), gdi_error_init(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_error_get – retrieve error information from the database connection

SYNOPSIS

```
#include "libgdi.h"

int
gdi_error_get (conn, errcode, errtext, maxtext, status, severity)
dbConn          *conn;          /* (i) database connection */
int             *errcode;       /* (o) specific error code */
char            *errtext;       /* (o) error text */
int             maxtext;        /* (i) length of errtext variable*/
int             *status;        /* (o) general status */
int             *severity;      /* (o) severity */
```

DESCRIPTION

Errors are reported on a connection by connection basis. gdi_error_get() retrieves error information from the database connector.

ARGUMENTS

conn The database connector. If NULL, global error information is retrieved.

errcode Specific error code.

errtext Message text for the error code.

maxtext Size of the *errtext* string, controlling how much text may be copied into the user's *errtext* variable.

status GDI_SUCCESS or GDI_FAILURE.

severity GDI_NOERROR, GDI_FATAL, or GDI_WARNING.

DIAGNOSTICS

gdi_error_get() always returns GDI_SUCCESS.

FILE

gdi_error.c

SEE ALSO

gdi_error_flags(3), **gdi_error_init**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_error_init – initialize error handling flags

SYNOPSIS

```
#include "libgdi.h"

int
gdi_error_init (conn, debug, threshold, reserved1, reserved2)
dbConn          *conn;          /* (i) database connection */
int             debug;          /* (i) GDI_DEBUG_OFF, GDI_DEBUG_ON, GDI_DEBUG_VERBOSE */
int             threshold;      /* (i) GDI_WARNING or GDI_FATAL */
int             reserved1;      /* not used */
int             reserved2;      /* not used */
```

DESCRIPTION

Errors are handled on a connection by connection basis. gdi_error_init() initializes the *debug* and *threshold* flags for a database connector. *debug* controls optional output of errors to *stderr*. *threshold* sets the level of error or warning that is treated as a failure by the GDI.

ARGUMENTS

conn The database connector. If NULL, sets global error flags and initializes global error indicators.

debug GDI_DEBUG_OFF (FALSE) by default. If set to GDI_DEBUG_ON (TRUE), errors are output automatically to *stderr*. If set to GDI_DEBUG_VERBOSE, non-error debug messages are output automatically to *stderr*.

threshold Sets the threshold at which an error or warning causes a GDI_FAILURE. A threshold of GDI_WARNING causes all warnings and errors to be treated as failures. A threshold of GDI_FATAL causes only *fatal errors to be treated as failures.*

reserved1 Reserved for future use.

reserved2 Reserved for future use.

DIAGNOSTICS

gdi_error_init() always returns GDI_SUCCESS.

FILE

gdi_error.c

SEE ALSO

gdi_error_flags(3), gdi_error_get(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_exit – close all open database connections

SYNOPSIS

```
#include "libgdi.h"

int
gdi_exit ()
```

DESCRIPTION

gdi_exit() closes all open database connections, freeing all database connection structures (*dbConn*).

DIAGNOSTICS

gdi_exit() always returns GDI_SUCCESS.

FILE

gdi_conn.c

SEE ALSO

gdi_close(3), **gdi_dead**(3), **gdi_open**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_flush – discard unprocessed query results

SYNOPSIS

```
#include "libgdi.h"

int
gdi_flush (conn, channo)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
```

DESCRIPTION

gdi_flush() dumps any unprocessed query results from the most recently executed query. 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 in case the user did not process all rows in the result set.

ARGUMENTS

conn The database connector for the connection the channel was opened on.

channo Channel to flush.

DIAGNOSTICS

gdi_flush() returns one of the following status values.

GDI_SUCCESS
Succeeded in flushing channel.

GDI_FAILURE
Flush failed; possibly the database connection dropped.

FILE

gdi_channel.c

SEE ALSO

gdi_abort(3)

AUTHOR

Jean T. Anderson, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_account – get database account name from database connector

SYNOPSIS

```
#include "libgdi.h"

int
gdi_get_account (conn, account, len)
dbConn          *conn;          /* (i) database connection */
char            *account;       /* (o) account name */
int             len;            /* (i) length of account argument */
```

DESCRIPTION

gdi_get_account() gets the database account name from the database connector.

ARGUMENTS

conn The database connector.

account Database account name is filled in by this routine.

len Length of the *account* argument.

DIAGNOSTICS

gdi_get_account() returns one of the following status values.

GDI_SUCCESS
Routine succeeded.

GDI_FAILURE
Not connected to database.

FILE

gdi_conn.c

SEE ALSO

gdi_get_database(3), **gdi_get_node**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_ArrayStructs – Get the results of a query in an array of structures.

SYNOPSIS

```
#include "libgdi.h"
#include "<type>_Astructs.h"

int
gdi_get_ArrayStructs (conn, query, array, maxrec, type)
dbConn              *conn;      /* (i) database connection */
char                *query;     /* (i) database query */
void                **array;    /* (o) array of structs */
int                 maxrec;     /* (i) maximum number of records to retrieve */
ArrayStructsArgs    *type;      /* (i) structure definition */
```

DESCRIPTION

gdi_get_ArrayStructs() submits a query to a database and returns the results in an array of structures. The array of structures is allocated by gdi_get_ArrayStructs(). It is the responsibility of the application to free the array. Headers containing a structure definition with fields matching the columns of the query are created with gdi_gen_Astructs(1).

The structure must contain a field for each column in the query however the columns need not be in the same order as the fields in the structure. The structure may contain more fields than those needed to match the query columns. The additional fields will be filled with default or "NA" values.

Note that the structure generated by gdi_gen_Astructs(1) matches the columns of a *query*, not the columns of a particular table. A query selecting a single column from a table or a query selecting columns from several tables may be used to generate the structure. The only restriction is that each column must be identified by a unique name.

ARGUMENTS

conn The database connector.

query The database query to be submitted to the database.

array The address of the array pointer to receive the query results. The results are allocated by gdi_get_ArrayStructs(). *Note: It is the responsibility of the application to free the structure.*

maxrec The maximum number of records, or tuples, to be returned from the database.

type A description of the array structure, the "NA" values and other information needed to process the results for output. The description is contained in the "<type>_Astructs.h" header.

EXAMPLE

The following example uses a header dumped by gdi_gen_Astructs(1) using the query, "select * from arrival". The structure definition in arrival_Astructs.h is shown below.

```
typedef struct arrival {
            char        sta [7];
            double      time;
            long        arid;
            long        jdate;
            long        stassid;
            long        chanid;
            char        chan [9];
            char        iphase [9];
            char        stype [2];
```

```
                    double      deltim;
                    double      azimuth;
                    double      delaz;
                    double      slow;
                    double      delslo;
                    double      ema;
                    double      rect;
                    double      amp;
                    double      per;
                    double      logat;
                    char        clip [2];
                    char        fm [3];
                    double      snr;
                    char        qual [2];
                    char        auth [16];
                    long        commid;
                    char        lddate [18];
        } Arrival;
```

The following code segment retrieves data from the database, displays the results, and then free's the result structure.

```
        #include "libgdi.h"
        #include "arrival_Astructs.h"

        ...

        dbConn      *conn;                              /* database connector */
        char        *query = "select * from arrival";
        Arrival     *tuples;                            /* tuples from the database */
        int         maxtup = 10;                        /* maximum number of tuples to return */
        int         ntuples;                            /* number of tuples returned */

        int         err_code;                           /* error handling variables */
        char        err_text [200];
        dbStatus    status;
        dbErrLev    severity;

        int         i;

        ... initialize the GDI and open a database connection ...

        if ((ntuples = gdi_get_ArrayStructs (conn, query, (void *) &tuples, maxtup,
                                &ARRIVAL_CONTAINER_DEF)) < 0)
        {
                    gdi_error_get (conn, &err_code, err_text, sizeof (errtext),
                                &status, &severity);

                    ... handle the error ...
        }

        for (i = 0; i < ntuples; i++)
        {
                    fprintf (stdout, "%6s %8s %.3f %10d %10.3f %s0,
                                tuples[i].sta, tuples[i].chan, tuples[i].time,
                                tuples[i].arid, tuples[i].azimuth, tuples[i].lddate);
        }
```

```
                free (tuples);
```

DIAGNOSTICS

gdi_get_ArrayStructs() returns the number of tuples retrieved if successful, otherwise it returns -1. Error codes and messages may be retrieved from the database connector with gdi_error_get(3).

FILE

gdi_ArrayStructs.c, gdi_ArrayStructs.h

SEE ALSO

gdi_add_ArrayStructs(3), gdi_error_get(3), gdi_gen_Astructs(1)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_counter – get unique database key(s)

SYNOPSIS

```
#include "libgdi.h"

int
gdi_get_counter (conn, tablename, keyname, nkeys, keyvalue)
dbConn          *conn;          (i) database connection
char            *tablename;     (i) name of key table
char            *keyame;        (i) name of key
int             nkeys;          (i) number of keys requested
long            *keyvalue;      (o) highest key value assigned
```

DESCRIPTION

gdi_get_counter() assigns unique sequential numbers to integer identifiers, called *keys*, in the database. It manages key assignment in the named table, which stores the name of the key in (*keyname*) and the last number assigned (*keyvalue*). Given the name of the key in *keyname*, gdi_get_counter () retrieves its value from the database, increments it by the amount in *nkeys*, writes it back to the database, and stores the result in *keyvalue* to be used by the calling application.

ARGUMENTS

conn	The database connector.
tablename	Name of the table used for dispensing key values.
keyname	Name of the key.
nkeys	Number of consecutive key values to assign.
keyvalue	Highest unique key value requested.

C EXAMPLES

The following example gets one *mesgid* key from the *lastid* table accessible by the current account:

```
#include "libgdi.h"

dbConn     *conn;

           /* variables for call to gdi_get_counter */
char       *tablename = "lastid";  /* name of key table */
char       *keyname = "mesgid";   /* name of key */
int        nkeys;                 /* number of keys to get */
int        keyval;                /* unique key value */

           /* error handling variables */
int        error_code, status, severity;
char       error_string [GDI_ERROR_SIZE + 1];

           ... open a database connection ...

keys=1;

if ((gdi_get_counter(conn, tablename, keyname, nkeys, &keyval)) != GDI_SUCCESS)
{
           gdi_error_get (conn, &error_code, error_string, sizeof(error_string),
                     &status, &severity);
           fprintf (stderr, "Error %d: '%s'\n", error_code, error_string);
           exit (GDI_FAILURE);
}
```

If no error occurred, *keyval* now contains one unique value the application may use.

If *nkeys* was 5, *keyval* would contain the highest of the 5 unique ids the application may use. For example, if *keyval* is 10, the application may use keys 6 through 10.

If *nkeys* was 0, *keyval* would contain the last value assigned--and the calling application should not use it since it was already used by another application.

DATABASE CONFIGURATION

The table must be created; for example:

SYBASE:

```
create table lastid (
          keyname        char(15)       not null,
          keyvalue       int            not null,
          lddate         datetime       null)
```

ORACLE:

```
create table lastid (
          keyname        varchar(15)    not null,
          keyvalue       number(8)      not null,
          lddate         date);
```

The *keyname* field contains the name of an integer primary or foreign key such as *mesgid*. The *keyvalue* field contains the last value which was used for the key in *keyname*. The *lddate* field contains the last time *keyname* was updated.

The table must be populated with the appropriate *keynames* for the database installation. The following examples demonstrate how to insert a new key and initialize it to 0:

SYBASE: insert into lastid (keyname, keyvalue, lddate) values ('mesgid', 0, getdate ())

ORACLE: insert into lastid (keyname, keyvalue, lddate) values ('arid', 0, sysdate);

The lastid table should be accessible to all who need to acquire keys:

```
grant select, update on lastid to public
```

NOTES

gdi_get_counter() explicitly commits the transaction on success, or rolls it back if an error occurs. Key values should be acquired before starting an SQL work group since the gdi_get_counter() is a work group in and of itself.

Currently there is no mechanism for recovering lost keys. For example, if an application gets a key value and the system goes down before the application has used the value, it will be lost.

DIAGNOSTICS

The following codes are returned from gdi_get_counter() to the calling application:

GDI_SUCCESS

This routine succeeded.

GDI_FAILURE

An error occurred. Specific error code and message may be retrieved with gdi_error_get().

FILE

gdi_get_counter.c

SEE ALSO

gdi_error_get (3)

AUTHOR

Jean Anderson, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_database – get database name from database connector

SYNOPSIS

```
#include "libgdi.h"

int
gdi_get_database (conn, database, len)
dbConn          *conn;          /* (i) database connection */
char            *database;      /* (o) database name */
int             len;            /* (i) length of database argument */
```

DESCRIPTION

gdi_get_database() gets the database name from the database connector.

ARGUMENTS

conn The database connector.

database Database name is filled in by this routine.

len Length of the *database* argument.

DIAGNOSTICS

gdi_get_database() returns one of the following status values.

GDI_SUCCESS

Routine succeeded.

GDI_FAILURE

Not connected to database.

FILE

gdi_conn.c

SEE ALSO

gdi_get_account(3), **gdi_get_node**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_dboption – Get the state of a database option

SYNOPSIS

```
#include "libgdi.h"

int
gdi_get_dboption (conn, channo, option, setting)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
dbOption        option;         /* (i) option to be set   */
char            *setting;       /* (o) value of the option */
int             int;            /* (i) length of 'setting' */
```

DESCRIPTION

The state of various database options may be retrieved by gdi_get_dboption(). Some options are set at the connection level, others at the channel level. Most options are specific to a database vendor. If the value is requested for an option which is not applicable to the vendor, setting is left untouched.

A database option may be set through gdi_get_dboption(3). Some options, such as GDI_PROC_C, are not settable but their states may still be retrieved.

ARGUMENTS

conn The database connector.

channo The channel number. *channo* is ignored by options that are set at the connector level.

option The option to be retrieved.

setting A char array in which the setting string will be stored.

len The length of the setting array.

OPTIONS

The following options may be retrieved:

GDI_VERSION
The version number of the GDI library.

GDI_AUTO_COMMIT
Oracle. "1" if auto commit is on, "0" if off. Auto commit is off by default. If auto commit is on, each database statement is automatically committed as soon as it is executed. If auto commit is off, database statements are treated as part of a transaction which is explicitly commited or rolled back with gdi_commit() or gdi_rollback().

GDI_PRO_C
Oracle. "1" if Pro*C mode is enabled, otherwise "0". The option applies to the entire connection. Pro*C is enabled by opening the connection using oracle_open(). The option can not be changed after the connection has been opened.

USAGE

The example below gets the setting of GDI_AUTO_COMMIT.

```
dbConn      *conn;
char        *setting;
int         len;

... initialize and open a connection ...

if (gdi_get_dboption (conn, GDI_DEFALUT_CHAN, GDI_AUTO_COMMIT,
            &setting, &len) != GDI_SUCCESS)
```

```
{
            ... handle error ...
}

printf ("Auto Commit = %s0, setting);
```

DIAGNOSTICS

gdi_get_dboption() returns one of the following status values:

GDI_SUCCESS
Operation succeeded.

GDI_FAILURE
Operation failed; possibly the connection dropped.

FILE

gdi_option.c

SEE ALSO

gdi_commit(3), **gdi_rollback**(3), **gdi_set_dboption**(3), **oracle_open**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_node – get database node name from database connector

SYNOPSIS

```
#include "libgdi.h"

int
gdi_get_node (conn, node, len)
dbConn          *conn;          /* (i) database connection */
char            *node;          /* (o) node name */
int             len;            /* (i) length of node argument */
```

DESCRIPTION

gdi_get_node() gets the database node name from the database connector.

ARGUMENTS

conn	The database connector.
node	Database node name is filled in by this routine.
len	Length of the *node* argument.

DIAGNOSTICS

gdi_get_node() returns one of the following status values.

GDI_SUCCESS
Routine succeeded.

GDI_FAILURE
Not connected to database.

FILE

gdi_conn.c

SEE ALSO

gdi_get_account(3), gdi_get_database(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_vendors – get a list of the vendors supported by GDI

SYNOPSIS

```
#include "libgdi.h"

char **
gdi_get_vendors ()
```

DESCRIPTION

gdi_get_vendors() returns a NULL terminated array of strings containing the names of the database vendors supported by the GDI.

SAMPLE CODE

```
char      **vendors;
int       i;

vendors = gdi_get_vendors ();

fprintf (stdout, "The supported GDI vendors are:\n");

for (i = 0; vendors[i] != NULL; i++)
                fprintf (stdout, "\t%s\n", vendors[i]);

fflush (stdout);
```

FILE

gdi_link.c

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_init – initialize the GDI

SYNOPSIS

```
#include "libgdi.h"

int
gdi_init (appname, gdihome)
char            *appname;       /* (i) application name*/
char            *gdihome;       /* (i) GDI home directory*/
```

DESCRIPTION

gdi_init() initializes the GDI.

ARGUMENTS

appname Application name (actual name of the executable).

gdihome Directory where GDI is installed. The GDI searches gdihome/lib for the GDI vendor interface libraries to be dynamically located. If gdi_init() has not been called or if gdihome is NULL or an empty string, "", then the GDI will use the environment variable, GDIHOME.

DIAGNOSTICS

gdi_init() returns one of the following status values.

GDI_SUCCESS
GDI successfully initialized.

GDI_FAILURE
Failure in initialization, possibly the application name was invalid.

FILE

gdi_link.c

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_insert – Insert data into a database table

SYNOPSIS

```
#include "libgdi.h"

int
gdi_insert (conn, table_name, datain)
dbConn          *conn;           /* (i) database connection */
char            *table_name;     /* (i) database table name */
dbObj           *datain;         /* (o) dbObj – data to be inserted */
```

DESCRIPTION

gdi_insert() inserts data into a database table. The data is contained in the tuples of the dbObj. The tuple constructor is used to access the data in the tuples. The column definitions in the dbObj are used to identify the columns of the database that are to receive the data.

Data is inserted using the fastest mode for the particular database. In the case of ORACLE, data is inserted using array inserts. SYBASE inserts use SYBASE's bulk copy mechanism.

ARGUMENTS

conn The database connector.

table_name The name of the table into which the data is to be inserted.

datain The dbObj containing the data to be inserted.

DIAGNOSTICS

gdi_insert() returns one of the following status values:

GDI_SUCCESS
Insert executed successfully.

GDI_FAILURE
Not connected to database or error executing command.

FILE

gdi_insert.c

SEE ALSO

gdi_submit(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_obj_create – allocate a new dbObj

SYNOPSIS

```
#include "libgdi.h"

dbObj*
gdi_obj_create (constr)
dbConstr          *constr;          /* (i) data constructor */
```

DESCRIPTION

gdi_obj_create() allocates a new *dbObj*. The constructor pointed to by *constr* is copied into the *dbObj* constructor field of the new *dbObj*. If gdi_obj_create() is successful, a pointer to the new *dbObj* is returned. NULL is returned if an error occurred.

The *dbObj* allocated should be accessed using the macros and functions provided by **libgdi.a**. Examples may be found in the test routine *libsrc/libgendb/test/tst_dbobj.c*.

ARGUMENTS

constr This is the tuple "constructor" which specifies pointers to functions that access the tuples in the *dbObj*. A default constructors is provided in *libgdi.h*. The GDI_DEFAULT constructor can be used when calling gdi_obj_create(), unless the user wants to specify a different tuple structure. Additional constructors include GDI_TURBO and GDI_SDI.

DIAGNOSTICS

gdi_obj_create() returns a pointer to the new *dbObj* if successful, or NULL if an error occurred.

FILE

gdi_dbobj.c

SEE ALSO

gdi_obj_destroy(3), **gdi_submit**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_obj_destroy – free memory allocated for a dbObj

SYNOPSIS

```
#include "libgdi.h"

int
gdi_obj_destroy (obj)
dbObj          *obj;          /* (i) database object */
```

DESCRIPTION

The *dbObj* is a generic structure containing database data, status and error information. A *dbObj* is normally created when a user calls a database access function, such as gdi_submit(). After extracting the information returned in the *dbObj*, the user should call gdi_obj_destroy() to free the memory allocated to the structure.

ARGUMENTS

obj A database object structure containing status, errors and other results of a database command.

DIAGNOSTICS

gdi_obj_destroy() always returns GDI_SUCCESS.

FILE

gdi_dbobj.c

SEE ALSO

gdi_obj_create(3), **gdi_submit**(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_open – establish a connection to the database

SYNOPSIS

```
#include "libgdi.h"

dbConn *
gdi_open (vendor, account, password, database, server, appname)
char            *vendor;        /* (i) database vendor  */
char            *account;       /* (i) database account */
char            *password;      /* (i) account password */
char            *database;      /* (i) database or machine */
char            *server;        /* (i) database server  */
char            *appname;       /* (i) application name  */
```

DESCRIPTION

Given the valid database connect information, gdi_open() opens a database connection to the specified database vendor, and creates and initializes the *dbConn* database connection structure.

More than one connection may be established, including a mix of database vendors. Two channels for each connection are opened. More channels may be opened with gdi_open_channel().

ARGUMENTS

Many of these parameters may be NULL depending on the database vendor.

vendor Required parameter. NULL-terminated string containing the name of the database vendor. *libgdi.h* includes string macros for each database supported (GDI_MONTAGE_S, GDI_ORACLE_S, GDI_POSTGRES_S, GDI_SYBASE_S). A GDI_ORACLE_PROC_S vendor option is also available, which establishes a pro*c connection to ORACLE. This allows programmers to link in pro*c routines.

account NULL-terminated string containing the database account or user name. ORACLE account names may include the password or the entire ORACLE Version 6 database connect string; for example, *gdidemo/gdidemo* or *gdidemo/gdidemo@t:skrymir:dev.*

password NULL-terminated string containing the account password. May be NULL for ORACLE if the *account* argument includes the password. May be NULL for other databases if a NULL password is allowed for the associated account.

database NULL-terminated string containing the database name for MONTAGE, POSTGRES, or SYBASE, or the SQL*Net connect string (*i.e.*, t:skrymir:dev) for ORACLE. May be NULL for ORACLE if the connect string is included in the *account* argument, or if either the TWO_TASK or ORACLE_SID environment variables are set. If NULL for all databases except ORACLE, the user's default database is opened.

server Name of the database server. May be NULL.

appname Application name (only used by SYBASE). May be NULL.

DIAGNOSTICS

If the attempt to open a connection fails, the *dbConn* returned will be NULL.

FILE

gdi_conn.c

SEE ALSO

gdi_close(3), gdi_dead(3), gdi_exit(3), gdi_get_account(3), gdi_get_database(3), gdi_get_node(3), gdi_get_vendors(3), gdi_open_channel(3), oracle_open(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_open_channel – open additional channel on a specified database connection

SYNOPSIS

```
#include "libgdi.h"

int
gdi_open_channel (conn, channo)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (o) channel number address  */
```

DESCRIPTION

A connection (dbConn) to the database may have multiple query channels. A channel is an MI_CONNECTION for MONTAGE, a cursor for ORACLE, a portal for POSTGRES, and a DBPROCESS for SYBASE. For example, at the time an ORACLE connection is established, two channels ("cursors") are automatically opened. gdi_open_channel() opens additional channels.

ARGUMENTS

conn The database connector for the connection on which to open the channel.

channo Channel number. The number gets filled in by this routine.

DIAGNOSTICS

gdi_open_channel() returns one of the following status values.

GDI_SUCCESS

Succeeded in opening channel.

GDI_FAILURE

Could not open channel.

FILE

gdi_channel.c

SEE ALSO

gdi_channel_is_open(3), gdi_close_channel(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_print_coldefs – output column definitions to stdout

SYNOPSIS

```
#include "libgdi.h"

int
gdi_print_coldefs (obj)
dbObj          *obj;          /* (i) database data object */
```

DESCRIPTION

gdi_print_coldefs() prints the column definitions of the database object, *dbObj*, to *stdout*. To print the *dbObj* use gdi_print_dbobj(). To print the actual data use gdi_print_tuples().

Column attributes printed are:

Name column name.

Null? is a database Null allowed for this column? 1 if Null is permitted. 0 if not.

Ctype integer values representing "C" language data types as defined in the include file libgdi.h, for example: M_INTEGER, M_STRING.

StrSize string length if column is a string type.

ArraySize array length if column is an array type.

Prec database precision value.

Scale database scale value.

Dbtype integer values representing database data types as defined in the libgdi.h. For ORACLE, the convention GDI_ORA_CHAR, GDI_ORA_NUMBER, etc. is used.

DbtypeStr human readable representation of the database type.

ARGUMENTS

obj The database data object.

DIAGNOSTICS

gdi_print_coldefs() returns one of the following status values.

GDI_SUCCESS

No problem outputting the column definitions.

GDI_FAILURE

NULL *dbObj* passed in.

FILE

gdi_print.c

SEE ALSO

gdi_print_conn(3), gdi_print_dbobj(3), gdi_print_tuples(3)

AUTHOR

Mari Mortell, SAIC Geophysical Systems Operation

NAME

gdi_print_conn – output the contents of the database connection structure to *stdout*

SYNOPSIS

```
#include "libgdi.h"

int
gdi_print_conn (conn)
dbConn          *conn;          /* (i) database connection */
```

DESCRIPTION

gdi_print_conn() prints the contents of the database connection structure, *dbConn*, to *stdout*. If a connection to a vendor has been made, the contents of the vendor specific connection are also printed.

ARGUMENTS

conn The database connector.

DIAGNOSTICS

gdi_print_conn() returns one of the following status values.

GDI_SUCCESS

No problem outputting *dbConn*.

GDI_FAILURE

NULL *dbConn* passed in.

FILE

gdi_print.c

SEE ALSO

gdi_print_dbobj(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_print_dbobj – output dbObj contents to stdout

SYNOPSIS

```
#include "libgdi.h"

int
gdi_print_dbobj (obj)
dbObj           *obj;           /* (i) obj */
```

DESCRIPTION

gdi_print_dbobj() outputs the contents of the database object, *dbObj*, to *stdout*. To print the column definitions use gdi_print_coldefs(). To print the actual data use gdi_print_tuples().

dbObj attributes printed are:

Affected Rows The number of rows affected by the database statement.

Tuples The number of rows of data stored in the dbObj.

Columns The number of columns in each row.

Status The return status of the database statement.

More Rows gdi_submit() allows a limit to be specified on the number of rows returned. "More Rows" is TRUE if more data exists in the database which satisfies the query than were returned.

Query The database statement.

ARGUMENTS

obj The database object.

DIAGNOSTICS

gdi_print_dbobj() returns one of the following status values.

GDI_SUCCESS
No problem outputting *dbObj*.

GDI_FAILURE
NULL *dbObj* passed in.

FILE

gdi_print.c

SEE ALSO

gdi_print_coldefs(3), **gdi_print_tuples**(3)

AUTHOR

Mari Mortell, SAIC Geophysical Systems Operation

NAME

gdi_print_tuples – print tuple data to stdout

SYNOPSIS

```
#include "libgdi.h"

int
gdi_print_tuples (dbobj, format, header)
dbObj           *dbobj;          /* (i) database object */
int             format;          /* (i) GDI_FIXED_SPACE or GDI_DELIMITED */
int             header;          /* (i) TRUE for column name headings, FALSE for data only */
```

DESCRIPTION

gdi_print_tuples() prints the tuple data in the database object, *dbObj*, to *stdout*. To print the *dbObj* use gdi_print_dbobj(). To print the column definitions use **gdi_print_coldefs()**.

Specifying GDI_FIXED_SPACE causes the tuples to be printed in tabular form. Numbers are right justified. Strings are left justified. GDI_DELIMITED, prints a comma without white space between fields. Strings and chars are enclosed in double quotes. This output was intended to be a flat file format compatible with a number of database vendors. The column name headings can be enabled or disabled.

ARGUMENTS

obj The database data object.

format GDI_FIXED_SPACE or GDI_DELIMITED.

header TRUE to enable the output of column name headings, FALSE for data only.

DIAGNOSTICS

gdi_print_tuples() returns one of the following status values.

GDI_SUCCESS
: No problem outputting tuples.

GDI_FAILURE
: NULL *dbObj* passed in.

FILE

gdi_print.c

SEE ALSO

gdi_print_coldefs(3), **gdi_print_dbobj**(3)

AUTHOR

Mari Mortell SAIC Geophysical Systems Operation

NAME

gdi_rollback – rollback current transaction

SYNOPSIS

```
#include "libgdi.h"

int
gdi_rollback (conn, channo, tran_name)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
char            *tran_name;     /* (i) transaction name */
```

DESCRIPTION

A database transaction is a statement, or statements, treated as an atomic unit. gdi_rollback() ends the current transaction and cancels all pending changes to the database.

Note that transaction management is implemented slightly differently in all the databases the gdi supports.

ARGUMENTS

conn The database connector.

channo The channel number (SYBASE and MONTAGE). SYBASE transactions are handled at the DBPROCESS level. MONTAGE transactions are handled at the database connection level, but each gdi query channel maps to a separate database connection. The channel argument is ignored for ORACLE and POSTGRES.

tran_name The transaction name of the transaction to be rolled back. This argument is only valid for SYBASE, which allows nested, named transactions.

DIAGNOSTICS

gdi_rollback() returns one of the following status values.

GDI_SUCCESS
Rollback succeeded.

GDI_FAILURE
Rollback failed; possibly the connection dropped.

FILE

gdi_tran.c

SEE ALSO

gdi_begin_tran(3), gdi_commit(3), gdi_savepoint(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_savepoint – set a savepoint

SYNOPSIS

```
#include "libgdi.h"

int
gdi_savepoint (conn, channo, sname)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
char            *sname;         /* (i) savepoint name */
```

DESCRIPTION

A database transaction is a statement, or statements, treated as an atomic unit. gdi_savepoint() identifies a point in a transaction to which a process can later rollback with the *rollback to savepoint savepoint_name* statement.

To rollback to a named savepoint, the process must build a text string containing the entire SQL statement, then execute the statement with a call to gdi_submit().

A call to gdi_rollback() or gdi_commit() negates all savepoints.

Transaction management is implemented slightly differently in all the databases the gdi supports.

ARGUMENTS

conn The database connector

channo Setting a savepoint involves a SQL command that must be executed on a channel. For SYBASE, it sets a savepoint only for activity on that channel since transactions are handled at the DBPROCESS level, not the database connection level. For ORACLE it sets a savepoint at the dbConn level because transactions are at the database connection level. MONTAGE and POSTGRES currently do not support savepoints.

DIAGNOSTICS

gdi_savepoint() returns one of the following status values.

GDI_SUCCESS

Savepoint succeeded.

GDI_FAILURE

Savepoint failed; possibly the connection dropped.

FILE

gdi_tran.c

SEE ALSO

gdi_commit(3), gdi_rollback(3), gdi_submit(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_set_dboption – Set or clear a database option

SYNOPSIS

```
#include "libgdi.h"

int
gdi_set_dboption (conn, channo, option, setting)
dbConn          *conn;          /* (i) database connection */
int             channo;         /* (i) channel number */
dbOption        option;         /* (i) option to be set    */
char            *setting;       /* (i) value to set option to */
```

DESCRIPTION

Various database options may be set by the application through gdi_set_dboption(). An option may be cleared or set to default be calling gdi_set_dboption() with a NULL setting. Some options are settable at the channel level.

Most options are specific to a database vendor. If an application attempts to set an option that is not applicable to the database, a warning is issued but otherwise the action is ignored.

The state of a database option may be ascertained through gdi_get_dboption(3). Some options, such as GDI_PRO_C, are not settable but their states may still be retrieved.

ARGUMENTS

conn The database connector.

channo The channel number. *channo* is ignored by options that are set at the connector level.

option The option to be set or cleared.

setting A string containing the value to set the option to. If *setting* is a NULL or empty string, the option is cleared or set to the default value.

OPTIONS

The following options may be set:

GDI_AUTO_COMMIT

Oracle. Set auto commit on or off ("1" or "0"). Auto commit is off by default and is set at the connection level. Setting auto commit on causes each database statement to be automatically committed as soon as it is executed.

GDI_CONFIG

Montage, Postgres. Checks for existence of GDI database support objects. If set to GDI_CONFIG_CHECK, returns GDI_FAILURE if objects do not exist. If set to GDI_CONFIG_INSTALL, tries to create the objects if they do not already exist. If set to GDI_CONFIG_REMOVE, removes GDI objects.

DIAGNOSTICS

gdi_set_dboption() returns one of the following status values:

GDI_SUCCESS

Operation succeeded.

GDI_FAILURE

Operation failed; possibly the connection dropped.

FILE

gdi_option.c

SEE ALSO

gdi_get_dboption(3)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_sleep – sleep a random number of seconds

SYNOPSIS

```
#include "libgdi.h"

void
gdi_sleep (max_sleep)
int                 max_sleep;        /* (i) maximum number of seconds to sleep */
```

DESCRIPTION

gdi_sleep() sleeps a random number of seconds that does not exceed *max_sleep* seconds. The sleep is random so processes pinging the same resource will become de-synchronized and retry at different times (used by gdi_get_counter(), for example).

ARGUMENTS

max_sleep The maximum number of seconds to ever sleep. If set to 0, does not sleep.

FILE

gdi_sleep.c

SEE ALSO

gdi_get_counter(3)

AUTHOR

Jean T. Anderson, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_submit – submit a database command

SYNOPSIS

```
#include "libgdi.h"

int
gdi_submit (conn, cmd_batch, max_records, constr, results)
dbConn          *conn;          /* (i) database connection */
char            *cmd_batch;     /* (i) database command(s) */
int             max_records;    /* (i) maximum number of records to fetch */
dbConstr        *constr;        /* (i) tuple constructor */
dbObj           **results;      /* (o) dbObj – status, errors, data */
```

DESCRIPTION

gdi_submit() sends a database command to the database to be executed. The results of the command, including status, errors, and tuples, if any, will be returned in the *results* structure.

The database commands must be written in the native language of the target database. The commands must be complete and syntactically correct.

For ORACLE database connections, the types of commands that may be executed include array fetches, inserts, updates and deletes without bind variables. DDL commands such as create, drop or alter table, commit, and rollback can also be done with gdi_submit(). Timeouts can occur while waiting for DDL locks.

Sample commands allowed for ORACLE and SYBASE connections include:

```
"select * from arrival"
"select sta, chan from arrival"
"select o.orid, a.arid, o.lat, o.lon, o.depth, o.time, a.phase,
     ar.time, ar.azimuth, ar.slow from assoc a, arrival ar,
     origin o where a.orid=o.orid and a.arid=ar.arid"
"select count(*) from origin, origerr"
"SELECT a.sta, a.time, b.wfid, a.lddate
      from atable a, dyn b where a.sta = b.sta"
"select max(sta), max(time), min(arid) from arrival where arid in
     (select arid from assoc where orid=3679)"
"update arrival set arid = 5 where arid = 7"
"delete from arrival where arid = 1234"
"select * from arrival where 1=2"       --> performs a describe
```

Sample ORACLE specific commands allowed include:

```
"select stddev(y) std_y from datamatrix"
"create table my_arrival as select * from arrival"
"insert into mytable (sta, time, wfid, lddate ) values( 'NRA0', 87654321.99, 1001,
     TO_DATE('19920527 17:21:59', 'YYYYMMDD HH24:MI:SS') )"
```

Sample SYBASE specific commands allowed include:

```
"select * into newtable from oldtable"      /* create table */
"insert into mytable (sta, time, wfid, lddate ) values ( 'NRA0', 87654321, 1001, getdate())"
"insert into mytable (lddate ) values ( 'Oct 15 1993  3:08:0' )"
"insert into mytable (lddate ) values ( 'Oct 15 1993  3:08:0PM' )"
```

Calculated columns should be named for SYBASE or the column name will be NULL. for example:
"select max(keyvalue) 'max key' from lastid"

For ORACLE Version 6 database connections, gdi_submit() automatically uses a default date mask, 'YYYYMMDD HH24:MI:SS', for columns with database type "date". For ORACLE Version 7, the date mask may be specified by the user. If a to_char() conversion is used for a date column, the column's datatype becomes "string" and is no longer recognized as a date.

After a command which changes the contents of the database completes successfully, ORACLE users should call ORACLE gdi_commit() to commit the transaction. The user is also responsible for calling gdi_obj_destroy() to free the memory allocated for *results*.

SQL commands requiring bind variables are not implemented for ORACLE or SYBASE. For example:

```
delete from table where id = :e
```

Other SQL and SQL*Plus commands not implemented are:

```
define
describe
@sqlscript
spool
set timing on
column format
list
```

Although gdi_submit() does not execute the describe command, descriptions of the attributes may be obtained in the column definitions of the *dbObj* structure resulting from the query below:

```
select * from table where 1=2
```

ARGUMENTS

conn The database connector.

cmd_batch A NULL terminated string containing any database command or, for SYBASE and MONTAGE, a batch of commands. For instance, insert commands of the form "insert into tables (list of values)" may be submitted using this function. Commands that select data from the database will be handled using array fetches for ORACLE. The data will be returned in the *results* argument.

max_records This specifies the maximum number of records that may be fetched from the database. All records will be fetched if *max_records* is set to -1. If *max_records* = 0, the default maximum MAXREC is returned. *max_records* only applies to fetches.

constr This is the tuple constructor, which specifies the functions that build the tuples for the *results* argument. Default constructors are provided in *libgdi.h*. The GDI_DEFAULT constructor can be used when calling gdi_submit(), unless the user wants to define different functions. Additional constructors include GDI_TURBO and GDI_SDI.

results A *dbObj* structure created by gdi_submit(). It contains status, errors and other results of the database command. If the database command resulted in data being fetched from the database, *results* also contains the database tuples. For SYBASE and MONTAGE, *results* may be a linked list of *dbObj*'s, one for each command in the command batch.

The fields in a *dbObj* are described below:

tuples This field is the pointer to the structure containing data tuples, if any.

n_tuples *n_tuples* is the number of tuples.

col_def This field is a pointer to a null terminated array of *dbColDef* structures, containing column definitions. There is one column definition structure for each column in the database query.

query This is a null terminated string containing the database query or command.

rows_affected This is the number of database rows affected by the query or command. In the case of a fetch, the number of rows affected is the same as the number of tuples fetched.

cmd_num When a block of multiple commands is submitted to gdi_submit(), *cmd_num* is the number of the command within the block. Initially, only SYBASE connections will handle multiple commands.

more_rows If a database command results in more rows than were requested by the value specified in *max_records*, this field indicates that additional data tuples are available.

constructor The *constructor* consists of function pointers and flags that specify the structure of the tuples and the tuple container.

next_obj When a block of commands is submitted to the database, a *dbObj* is associated with each command. *next_obj* points to the *dbObj* corresponding to the next command in the block.

prev_obj *prev_obj* points to the *dbObj* corresponding to the previous command in a command block.

The information and fields in a *dbObj* should never be accessed directly. The GDI provides macros and functions to access the data.

The following macros are provided:

GDI_OBJ_NUM_TUPLES Get the number of tuples in a *dbObj.*

GDI_OBJ_ROWS_AFFECTED Get the number of rows affected by the command in a *dbObj.*

GDI_OBJ_QUERY Get the database query in a *dbObj.*

GDI_OBJ_CMD_NUM Get the command number with the command batch.

GDI_OBJ_MORE_ROWS Get the *more rows* flag from a *dbObj.*

GDI_OBJ_STATUS Get the command status from a *dbObj.*

GDI_OBJ_TUPLES Get the tuple container structure from a a *dbObj.*

GDI_OBJ_CONSTRUCTOR Get the pointer to the tuple constructor.

GDI_OBJ_COL_DEFS Get the pointer to the array of column definitions.

GDI_OBJ_COL_DEF Get the pointer to a specified column definition, given the column number in the command.

GDI_OBJ_COL_NAME Get the name of a column in a *dbObj,* given the column number within the command.

GDI_OBJ_COL_CTYPE Get the C type of a column in a *dbObj,* given the column number within the command.

GDI_OBJ_COL_PRECISION Get the database precision of a column in a *dbObj,* given the column number within the command. Precision is only valid for ORACLE data.

GDI_OBJ_COL_SCALE Get the database scale of a column in a *dbObj,* given the column number within the command. Scale is only valid for ORACLE data.

GDI_OBJ_COL_MAX_STRLEN Get the maximum length of a string column in a *dbObj*, given the column number within the command.

GDI_OBJ_COL_MAX_ARRLEN Get the maximum length of an array column in a *dbObj*, given the column number within the command. Array columns are only created by POSTGRES queries.

GDI_OBJ_COL_DBTYPE_S Get the string representation of the database type of a column in a *dbObj*, given the column number within the command.

GDI_OBJ_ALLOW_NULL Get the *allow_null* flag or a column, given the column number in the command.

The functions provided include:

gdi_obj_num_columns() Calculate the number of columns in a *dbObj*. Returns number of columns if successful, -1 if failure.

gdi_obj_value() Return a pointer to a database value, given a *dbObj*, a tuple number and a column number. The application must cast the pointer to the correct C type to access the data.

gdi_obj_find_value Return a pointer to a database value, given a *dbObj*, a tuple number and the column name instead of the column number.

gdi_obj_col_find_col_def() Return the number of a column in a *dbObj*, given the column name.

gdi_obj_col_num() Return the definition of a column in a *dbObj*, given the column name.

DIAGNOSTICS

gdi_submit() returns one of the following status values:

GDI_SUCCESS
Command executed successfully.

GDI_FAILURE
Not connected to database or error executing command.

FILE

gdi_submit.c

NOTES

Multiple command batches are not implemented yet for MONTAGE and SYBASE.

SEE ALSO

gdi_commit(3), **gdi_obj_destroy**(3), **gdi_print_coldefs**(3), **gdi_print_dbobj**(3), **gdi_print_tuples**(3)

AUTHOR

B. MacRitchie, Mari Mortell, K. Garcia, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_trace – turn database tracing on or off

SYNOPSIS

```
#include "libgdi.h"

int
gdi_trace (dbconn, state, filename)
dbConn          *conn           /* (i) database connector */
int                             state/* (i) TRUE or FALSE */
char                            *filename/* (i) name of file */
```

DESCRIPTION

gdi_trace() enables or disables database tracing. If the database connection is to a SYBASE database, the traces are dumped to a file specified by *filename*.

ARGUMENTS

conn The database connector.

state TRUE to turn tracing on, FALSE to turn tracing off.

filename Output filename (SYBASE only). May be a null or empty string, "".

DIAGNOSTICS

gdi_trace() returns one of the following status values.

GDI_SUCCESS
Trace successfully enabled or disabled.

GDI_FAILURE
gdi_trace() failed; possibly the connection dropped.

FILE

gdi_trace.c

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

ora_sqlca_error – stores SQLCA error in the database connector

SYNOPSIS

```
#include "libgdi.h"
#include "ora_proC.h"

int
ora_sqlca_error (conn, ptr_sqlca, str)
dbConn          *conn;          /* (i) database connection */
struct sqlca    *ptr_sqlca;     /* (i) SQLCA */
char            *str;           /* custom string */
```

DESCRIPTION

ora_sqlca_error() stores the status of a SQL statement executed by a PRO*C call based on the contents of the SQL Communication area (SQLCA). The database connection must be opened by oracle_open() to execute PRO*C routines.

ARGUMENTS

conn	The database connector.
ptr_sqlca	Pointer to the SQLCA.
str	Customized error string.

FILE

gdi_error.c

NOTES

Note that this is an ORACLE-specific routine highlighted here for users who wish to link their own PRO*C routines with libgdi.a.

SEE ALSO

oracle_open(3)

AUTHOR

Jean T. Anderson, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_close – close the specified database connection

SYNOPSIS

```
#include "gdi_f77.h"

integer function gdi_close (conn)
integer          conn          (i) database connection
```

DESCRIPTION

gdi_close() closes a connection to the database and frees the database connection structure, *dbConn*, associated with the *conn* parameter.

ARGUMENTS

conn The database connection handle of the connection to be closed.

DIAGNOSTICS

gdi_close() returns one of the following status values.

GDI_SUCCESS
Connection successfully closed.

GDI_FAILURE
Not connected to database.

FILE

gdi_f77_conn.c

SEE ALSO

gdi_open(3), gdi_open(3f)

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_error_get – retrieve error information from the database connection

SYNOPSIS

#include "gdi_f77.h"

subroutine gdi_error_get (conn, errcode, errtext, maxtext, status, severity)

integer	**conn**	(i) database connection
integer	**errcode**	(o) specific error code
character	**errtext**	(o) error text
integer	**maxtext**	(i) length of errtext variable
integer	**status**	(o) general status
integer	**severity**	(o) severity

DESCRIPTION

gdi_error_get() retrieves error information from the database connector.

ARGUMENTS

conn The database connection handle. If the handle is set to DB_NOCONN, then global error information is retrieved.

errcode Error code.

errtext Message text for the error code.

maxtext Size of the *errtext* string, controls how much text may be copied into the user's *errtext* variable.

status GDI_SUCCESS or GDI_FAILURE.

severity GDI_NOERROR, GDI_FATAL, or GDI_WARNING.

SAMPLE CODE

See test stubs in libsrc/libgendb/test/{oracle | postgres}.

FILE

gdi_f77_error.c

SEE ALSO

gdi_error_get(3), gdi_error_init(3f)

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_error_init – initialize error handling flags

SYNOPSIS

```
#include "gdi_f77.h"

subroutine gdi_error_init (dbconn, debug, threshold, reserved1, reserved2)
integer         dbConn          (i) database connection
integer         debug           (i) GDI_DEBUG_OFF, GDI_DEBUG_ON, GDI_DEBUG_VERBOSE
integer         threshold       (i) GDI_WARNING or GDI_FATAL
integer         reserved1       (i) not used
integer         reserved2       (i) not used
```

DESCRIPTION

Errors are handled on a connection by connection basis. gdi_error_init() initializes the *debug* and *threshold* flags for a database connector. *debug* controls optional output of errors to *stderr*. *threshold* sets the level of error or warning that is treated as a failure by the GDI.

ARGUMENTS

conn The database connection handle.

debug GDI_DEBUG_OFF (FALSE) by default. If set to GDI_DEBUG_ON (TRUE), errors are output automatically to *stderr*. If set to GDI_DEBUG_VERBOSE, non-error debug messages are output automatically to *stderr*.

threshold Sets the threshold at which an error or warning causes a GDI_FAILURE. A threshold of GDI_WARNING causes all warnings and errors to be treated as failures. A threshold of GDI_FATAL causes only fatal errors to be treated as failures.

reserved1 Reserved for future use.

reserved2 Reserved for future use.

FILE

gdi_f77_error.c

SEE ALSO

gdi_error_get(3f), gdi_error_init(3)

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_account – get database account name from database connector

SYNOPSIS

```
#include "gdi_f77.h"

int
gdi_get_account (conn, account)
dbConn          *conn;          /* (i) database connection */
char            *account;       /* (o) account name */
```

DESCRIPTION

gdi_get_account() gets the database account name from the database connector.

ARGUMENTS

conn The database connection handle.

account Database account name is filled in by this routine.

DIAGNOSTICS

gdi_get_account() returns one of the following status values.

GDI_SUCCESS
Routine succeeded.

GDI_FAILURE
Not connected to database.

FILE

gdi_f77_conn.c

SEE ALSO

gdi_get_database(3f), gdi_get_node(3f)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_database – get database name from database connector

SYNOPSIS

```
#include "gdi_f77.h"

int
gdi_get_database (conn, database)
dbConn          *conn;          /* (i) database connection */
char            *database;      /* (o) database name */
```

DESCRIPTION

gdi_get_database() gets the database name from the database connector.

ARGUMENTS

conn The database connection handle.

database Database name is filled in by this routine.

DIAGNOSTICS

gdi_get_database() returns one of the following status values.

GDI_SUCCESS
Routine succeeded.

GDI_FAILURE
Not connected to database.

FILE

gdi_f77_conn.c

SEE ALSO

gdi_get_account(3f), gdi_get_node(3f)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_get_node – get database node name from database connector

SYNOPSIS

```
#include "gdi_f77.h"

int
gdi_get_node (conn, node)
dbConn          *conn;          /* (i) database connection */
char            *node;          /* (o) node name */
```

DESCRIPTION

gdi_get_node() gets the database node name from the database connector.

ARGUMENTS

conn The database connection handle.

node Database node name is filled in by this routine.

DIAGNOSTICS

gdi_get_node() returns one of the following status values.

GDI_SUCCESS
Routine succeeded.

GDI_FAILURE
Not connected to database.

FILE

gdi_f77_conn.c

SEE ALSO

gdi_get_account(3f), gdi_get_database(3f)

AUTHOR

B. MacRitchie, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_init – initialize the GDI

SYNOPSIS

```
#include "gdi_f77.h"

integer function gdi_init (appname, gdihome)
character          appname          (i) application name
character          gdihome;         /* (i) GDI home directory*/
```

DESCRIPTION

gdi_init() initializes the GDI.

ARGUMENTS

appname Application name (actual name of the executable).

gdihome Directory where GDI is installed. The GDI searches gdihome/lib for the GDI vendor interface libraries to be dynamically located. If gdi_init() has not been called or if gdihome is an empty string, "", then the GDI will use the environment variable, GDIHOME.

DIAGNOSTICS

gdi_init() returns one of the following status values.

GDI_SUCCESS
GDI successfully initialized

GDI_FAILURE
Failure in initialization, possibly the application name was invalid.

FILE

gdi_link.c

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_map – manage relationships between FORTRAN data and gdi data

SYNOPSIS

```
#include "gdi_f77.h"

integer function gdi_open_map (conn)
integer          conn             (i) database connection

subroutine gdi_close_map (conn, map)
integer          conn             (i) database connection
integer          map_id           (i) map to close

subroutine gdi_destroy_map (conn, map)
integer          conn             (i) database connection
integer          map_id           (i) map to destroy

integer function gdi_add_map_field (conn, map, column_name, data_addr, data_type, string_len, array_len)
integer          conn             (i) database connection
integer          map_id           (i) map to add column to.
character        column_name      (i) name of the database column
integer          data_addr        (i) name of the destination FORTRAN array
integer          data_type        (i) data type of destination array
integer          string_len       (i) length of destination string
integer          array_len        (i) length of destination array
```

DESCRIPTION

The GDI Map functions allow the application to build a Map which contains a description of the FORTRAN output variables for the data returned from a database query. Each column in the query is mapped to a FORTRAN array on a one-to-one basis. The application builds a Map and then passes the Map ID to gdi_submit() along with the database query. gdi_submit() fills the FORTRAN output arrays as specified by the Map. Each query that returns data requires a valid Map. Multiple maps may be created. Maps may be reused by subsequent queries. When the Map is no longer needed, it may be destroyed.

gdi_open_map() begins a mapping reference.

gdi_close_map() ends a mapping reference.

gdi_destroy_map() deallocates the memory that the GDI allocated when the map was built. Data in the FORTRAN arrays are not affected.

gdi_add_map_field() adds an element, a reference to a FORTRAN output array and a query column, to a map.

ARGUMENTS

conn The database connection handle.

map_id Identifies the map to use in the operation. Multiple maps may be defined.

column_name The name of the database column from which data will be read.

data_addr The FORTRAN variable which will hold the retrieved data.

data_type The data type that the data_addr variable is.

string_len Describes how long each string is (should the column be a string column). If the data_type is not GDI_STRING, then this parameter should be zero (0).

array_len For ORACLE, this variable has no meaning and should always be zero (0). For POSTGRES, this variable indicates the number of rows in an array fetch.

DIAGNOSTICS

The Map functions return one of the following status values:

GDI_SUCCESS

The requested operation was performed.

GDI_FAILURE

The requested operation could not be performed. Use gdi_error_get() to get error information.

FILE

gdi_f77_map.c

SEE ALSO

gdi_error_get(3f)

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_open – establish a connection to the database

SYNOPSIS

```
#include "gdi_f77.h"

integer function gdi_open (vendor, account, password, database, server, appname)
character       vendor          (i) database vendor
character       account         (i) database account
character       password        (i) account password
character       database        (i) database or machine
character       server          (i) database server
character       appname         (i) application name
```

DESCRIPTION

gdi_open() opens a database connection to the specified database vendor. More than one connection may be established, including a mix of database vendors.

ARGUMENTS

Many of these parameters may be NULL depending on the database vendor.

vendor Required parameter. Character string containing the name of the database vendor. Currently supported vendors are "montage", "oracle", "postgres", and "sybase".

account Character string containing the database account or user name. ORACLE account names may include the password or the entire ORACLE Version 6 database connect string; for example, *gdidemo/gdidemo* or *gdidemo/gdidemo@t:skrymir:dev*.

password Character string containing the account password. May be an empty string, "", for ORACLE if the *account* argument includes the password.

database Character string containing the database for MONTAGE, POSTGRES, or SYBASE or the SQL*Net connect string (*i.e.*, t:skrymir:dev) for ORACLE. May be an empty string, "", for ORACLE if the connect string is included in the *account* argument, or if either the TWO_TASK or ORACLE_SID environment variables are set. If an empty string for all databases but ORACLE, the user's default database is opened.

server Name of the database server. Optional.

appname Application name (only used by SYBASE).

DIAGNOSTICS

If the attempt to open a connection fails, the database connection handle, *conn*, will be GDI_NOCONN.

FILE

gdi_f77_conn.c

SEE ALSO

gdi_close(3f), gdi_open(3)

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_submit – submit a database command

SYNOPSIS

```
#include "gdi_f77.h"

integer
gdi_submit (conn, map_id, cmd_batch, max_records, rows_retrieved, rows_affected, more_data)
integer      conn            (i) database connection
integer      map_id          (i) map id
character    cmd_batch       (i) string containing SQL command(s)
integer      max_records     (i) maximum number of records to fetch
integer      row_retrieved   (o) # of rows retrieved
integer      row_affected    (o) # of rows affected
logical      more_data       (o) signals more data in the database
```

DESCRIPTION

After a connection has been made to a database with **gdi_open()**, gdi_submit() sends a database command to the database to be executed. Data will be returned as described by the *map_id*.

The database commands must be written in the native language of the target database. The commands must complete and syntactically correct.

For ORACLE database connections, the types of commands that may be executed include array fetches, inserts, updates and deletes without bind variables. DDL commands such as create, drop or alter table, commit, and rollback can also be done with gdi_submit(). Timeouts can occur while waiting for DDL locks.

Sample commands allowed for ORACLE connections include:

```
"select * from arrival"
"select sta, chan from arrival"
"select o.orid, a.arid, o.lat, o.lon, o.depth, o.time, a.phase,
     ar.time, ar.azimuth, ar.slow from assoc a, arrival ar,
     origin o where a.orid=o.orid and a.arid=ar.arid"
"select stddev(y) std_y from datamatrix"
"select count(*) from origin, origerr"
"SELECT a.sta, a.time, b.wfid, a.lddate
      from atable a, dyn b where a.sta =b.sta"
"select max(sta), max(time), min(arid) from arrival where arid in
     (select arid from assoc where orid=3679)"
"create table my_arrival as select * from arrival"
"delete from arrival where arid = 1234"
"select * from arrival where 1=2"        --> performs a describe
```

For ORACLE Version 6 database connections, gdi_submit() automatically uses a default date mask, 'YYYYMMDD HH24:MI:SS', for columns with database type "date". For ORACLE Version 7, the date mask may be specified by the user. If a to_char() conversion is used for a date column, the column's datatype becomes "string" and is no longer recognized as a date.

After a command which changes the contents of the database completes successfully, the user should call gdi_commit() to commit the transaction.

ARGUMENTS

conn The database connection handle, returned from gdi_open().

cmd_batch A character string containing a database command. Any data fetched from the database will be placed in FORTRAN variables specified by the *map_id*. While the gdi C interface supports executing multiple commands in the cmd_batch, the FORTRAN interface does not. It is up to the programmer to ensure that only one command is executed at a time.

max_records This specifies the maximum number of records that may be fetched from the database. All records will be fetched if *max_records* is set to -1. If *max_records* = 0, the default maximum MAXREC is returned. *max_records* only applies to fetches.

map_id This identifies a description of the data variables in FORTRAN space.

rows_affected This is the number of database rows affected by the query or command. In the case of a fetch, the number of rows affected is the same as the number of tuples fetched.

rows_retrieved This is the number of database rows retrieved by the query or command. In the case of a fetch, the number of rows affected is the same as the number of tuples fetched.

more_rows If a database command results in more rows than were requested by the value specified in *max_records*, this field indicates that additional data tuples are available.

DIAGNOSTICS

gdi_submit() returns one of the following status values. Error codes and messages may be retrieved with gdi_error_get().

GDI_SUCCESS

Command executed successfully.

GDI_FAILURE

Not connected to database or error executing command.

FILE

gdi_f77_submit.c

SEE ALSO

gdi_error_get(3f), gdi_map(3f), gdi_open(3f), gdi_submit(3)

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

NAME

gdi_trace – turn database tracing on or off

SYNOPSIS

```
#include "gdi_f77.h"

subroutine gdi_trace (conn, state, filename)
integer         conn            (i) database connector
integer         state           (i) .TRUE. or .FALSE.
character       filename        (i) name of file
```

DESCRIPTION

gdi_trace() enables or disables database tracing. If the database connection is to a SYBASE database, the traces are dumped to a file specified by *filename*.

ARGUMENTS

conn The database connection handle.

state TRUE to turn tracing on, FALSE to turn tracing off.

filename Output filename (SYBASE only). May be null, i.e. ''.

SAMPLE CODE

See test stubs in libsrc/libgendb/test.

FILE

gdi_f77_trace.c

AUTHOR

H. Turner, SAIC Geophysical Systems Operation, Open Systems Division

Part V: Appendices

Appendix A. Bibliography

The following bibliography contains SQL references.

Emerson, Sandra L., Marcy Darnovsky and Judith S. Bowman, *The Practical SQL Handbook,* Reading, MA: Addison-Wesley Publishing Company, 1989.

This contains an excellent introduction to relational databases, relational database design, and the SQL language, with an emphasis on Sybase Transact-SQL.

Hursch, Carolyn J. and Jack L. Hursch, SQL, *The Structured Query Language,* Blue Ridge Summit, PA: TAB Books, Inc., 1988.

This introduces SQL to the novice.

van der Lans, Rick. F., *Introduction to SQL,* Reading, MA: Addison-Wesley Publishing Company, 1988.

This introduction to SQL is formulated around the creation of a sports club database. It is geared for the novice with a focus on ANSI SQL standard queries.

van der Lans, Rick. F., *The SQL Standard: A Complete Reference,* Hertfordshire, England: Prentice Hall International (UK) Ltd, 1988.

This reference is a companion guide to van der Lans' *Introduction to SQL.* It is much more readable than the ANSI X3.135-1986 document.

Appendix B. Data Types

The interface provides default conversions between database data types and C types. The tables below show the defaults for database to C and for C to database conversions. The defaults may be overridden by the application by manipulating the column definition in the Database Object (col_def in dbObj).

Table 16. Default Data Conversion - Database Types to C Types

Oracle(p,s)	Sybase	Ingres	C Types
	TINYINT		integer
NUMBER(<=5)	SMALLINT		integer
NUMBER(>5)	INT		long
NUMBER(x,>0)			double
NUMBER			double
FLOAT(<=24)	REAL		float
FLOAT(>24)	FLOAT		double
VARCHAR	VARCHAR		string
CHAR(>1)	CHAR (> 1)		string
CHAR(1)	CHAR (1)		char
DATE	DATETIME		string
	SMALLDATETIME		string
	MONEY		double
	SMALLMONEY		float
ROWID			long
	TIMESTAMP		
	SYSNAME		string
	BIT		integer
LONG			
	BINARY		
	VARBINARY		
RAW	TEXT		string
LONG RAW	IMAGE		

Table 17. Default Data Conversion - C Types to Database Types

C Types	Oracle(p,s)	Sybase	Ingres
integer	NUMBER (5)	INT	
long	NUMBER (10)	INT	
float	FLOAT (24)	REAL	
double	FLOAT (53)	FLOAT	
string [x<=256]	VARCHAR (x-1)	VARCHAR (x-1)	
string [x>256]		TEXT (x-1)	
char	CHAR (1)	CHAR (1)	

DISTRIBUTION LIST

RECIPIENT	NUMBER OF COPIES
DEPARTMENT OF DEFENSE	
ARPA/NMRO ATTN: Dr. R. Alewine, Dr. S. Bratt, and Dr. A. Ryall, Jr. 3701 North Fairfax Drive Arlington, VA 22203-1714	3
ARPA, OASB/Library 3701 North Fairfax Drive Arlington, VA 22203-1714	1
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
DEPARTMENT OF THE AIR FORCE	
AFTAC/TT ATTN: Dr. L. Himes, Dr. F. Pilotte, and Dr. D. Russell 130 South Highway A1A Patrick AFB, FL 32925-3002	3
AFTAC/TT, Center for Seismic Studies ATTN: Dr. R. Blandford 1300 North 17th Street, Suite 1450 Arlington, VA 22209-2308	1
Phillips Laboratory/GPEH ATTN: Mr. J. Lewkowicz 29 Randolph Road Hanscom AFB, MA 01731-3010	1
DEPARTMENT OF ENERGY	
Department of Energy ATTN: Dr. M. Denny Office of Arms Control Washington, D.C. 20585	1

Lawrence Livermore National Laboratory 4
ATTN: Dr. J. Hannon, Dr. K. Nakanishi, Dr. H. Patton,
and Dr. D. Springer
University of California
P.O. Box 808
Livermore, CA 94550

Los Alamos National Laboratory 1
ATTN: Dr. S. Taylor
P.O. Box 1663, Mail Stop C335
Los Alamos, NM 87545

Sandia National Laboratory 2
ATTN: Dr. E. Chael and Dr. M. Sharp
Division 9241
Albuquerque, NM 87185

OTHER GOVERNMENT AGENCIES

Central Intelligence Agency 1
ATTN: Dr. L. Turnbull
CIA-OSWR/NED
Washington, D.C. 20505

U.S. Geological Survey 1
ATTN: Dr. A. McGarr
Mail Stop 977
Menlo Park, CA 94025

U.S. Geological Survey 1
ATTN: Dr. W. Leith
Mail Stop 928
Reston, VA 22092

U.S. Geological Survey 1
ATTN: Dr. R. Masse
Denver Federal Building
Box 25046, Mail Stop 967
Denver, CO 80225

UNIVERSITIES

Boston College 1
ATTN: Dr. A. Kafka
Department of Geology and Geophysics
Chestnut Hill, MA 02167

California Institute of Technology 1
ATTN: Dr. D. Helmberger
Seismological Laboratory
Pasadena, CA 91125

Columbia University 2
ATTN: Dr. P. Richards and Dr. L. Sykes
Lamont-Doherty Geological Observatory
Palisades, NY 10964

Cornell University 1
ATTN: Dr. M. Barazangi
Institute for the Study of the Continent
Ithaca, NY 14853

IRIS, Inc. 2
ATTN: Dr. D. Simpson and Dr. G. van der Vink
1616 North Fort Myer Drive, Suite 1050
Arlington, VA 22209

Massachusetts Institute of Technology 1
ATTN: Dr. T. Jordan
Department of Earth, Atmospheric and Planetary Sciences
Cambridge, MA 02139

Massachusetts Institute of Technology 1
ATTN: Dr. N. Toksoz
Earth Resources Laboratory
42 Carleton Street
Cambridge, MA 02142

MIT Lincoln Laboratory, M-200B 1
ATTN: Dr. R. Lacoss
P.O. Box 73
Lexington, MA 02173-0073

San Diego State University 1
ATTN: Dr. S. Day
Department of Geological Sciences
San Diego, CA 92182

Southern Methodist University 2
ATTN: Dr. E. Herrin and Dr. B. Stump
Institute for the Study of Earth and Man
Geophysical Laboratory
Dallas, TX 75275

Southern Methodist University 1
ATTN: Dr. Gary McCartor
Department of Physics
Dallas, TX 75275

State University of New York at Binghamton 2
ATTN: Dr. J. Barker and Dr. F. Wu
Department of Geological Sciences
Vestal, NY 13901

St. Louis University 2
ATTN: Dr. R. Herrmann and Dr. B. Mitchell
Department of Earth and Atmospheric Sciences
St. Louis, MO 63156

The Pennsylvania State University 2
ATTN: Dr. S. Alexander and Dr. C. Langston
Geosciences Department
403 Deike Building
University Park, PA 16802

University of Arizona 1
ATTN: Dr. T. Wallace
Department of Geosciences, Building #77
Tucson, AZ 85721

University of California, Berkeley 2
ATTN: Dr. L. Johnson and Dr. T. McEvilly
Seismographic Station
Berkeley, CA 94720

University of California, Davis 1
ATTN: Dr. R. Shumway
Division of Statistics
Davis, CA 95616

University of California, San Diego 5
ATTN: Dr. J. Berger, Dr. L. Burdick, Dr. H. Given, Dr. B. Minster, and Dr. J. Orcutt
Scripps Institute of Oceanography, A-025
La Jolla, CA 92093

University of California, Santa Cruz 1
ATTN: Dr. T. Lay
Institute of Tectonics
Earth Science Board
Santa Cruz, CA 95064

University of Colorado 2
ATTN: Dr. C. Archambeau and Dr. D. Harvey
CIRES
Boulder, CO 80309

University of Connecticut 1
ATTN: V. Cormier
Department of Geology and Geophysics
U-45, Room 207
Storrs, CT 06268

University of Southern California 1
ATTN: Dr. K. Aki
Center for Earth Sciences
University Park
Los Angeles, CA 90089-0741

University of Wisconsin-Madison 1
ATTN: Dr. C. Thurber
Department of Geology and Geophysics
1215 West Dayton Street
Madison, WS 53706

DEPARTMENT OF DEFENSE CONTRACTORS

Center for Seismic Studies 3
ATTN: Dr. R. Bowman, Dr. J. Carter, and Dr. R. Gustafson
1300 North 17th Street, Suite 1450
Arlington, VA 22209

ENSCO, Inc. 2
ATTN: Dr. D. Baumgardt and Dr. Z. Der
5400 Port Royal Road
Springfield, VA 22151-2388

ENSCO, Inc. 2
ATTN: Dr. R. Kemerait and Dr. D. Taylor
445 Pineda Court
Melbourne, FL 32940-7508

Mission Research Corporation 1
ATTN: Dr. M. Fisk
735 State Street
PO Drawer 719
Santa Barbara, CA 93102-0719

Radix Systems, Inc. 1
ATTN: Dr. J. Pulli
201 Perry Parkway
Gaithersburg, MD 20877

Science Horizons 1
ATTN: Dr. T. Cherry
710 Encinitas Blvd., Suite 200
Encinitas, CA 92024

S-CUBED, 2
A Division of Maxwell Laboratory
ATTN: Dr. T. Bennett and Mr. J. Murphy
11800 Sunrise Valley Drive, Suite 1212
Reston, VA 22091

S-CUBED, 2
A Division of Maxwell Laboratory
ATTN: Dr. K. McLaughlin and Dr. J. Stevens
P.O. Box 1620
La Jolla, CA 92038-1620

SRI International 2
ATTN: Dr. A. Florence and Dr. S. Miller
333 Ravenswood Avenue, Box AF116
Menlo Park, CA 94025-3493

Teledyne Geotech 1
ATTN: Mr. W. Rivers
314 Montgomery Street
Alexandria, VA 22314-1581

TASC, Inc. 1
ATTN: Dr. R. Comer
55 Walkers Brook Drive
Reading, MA 01867

NON-US RECIPIENTS

Blacknest Seismological Center 1
ATTN: Dr. P. Marshall
UK Ministry of Defense
Blacknest, Brimpton
Reading FG7-FRS, UNITED KINGDOM

Institute for Geophysik 1
ATTN: Dr. H.-P. Harjes
Ruhr University/Bochum
P.O. Box 102148
4630 Bochum 1, GERMANY

NTNF/NORSAR 2
ATTN: Dr. S. Mykkeltveit and Dr. F. Ringdal
P.O. Box 51
N-2007 Kjeller, NORWAY

Societe Radiomana 1
ATTN: Dr. B. Massinon
27 Rue Claude Bernard
75005 Paris, FRANCE

University of Cambridge 1
ATTN: Dr. K. Priestley
Bullard Labs, Department of Earth Sciences
Madingley Rise, Madingley Road
Cambridge CB3, OEZ, ENGLAND

University of Toronto 1
ATTN: Dr. K.-Y. Chun
Geophysics Division
Physics Department
Ontario, CANADA