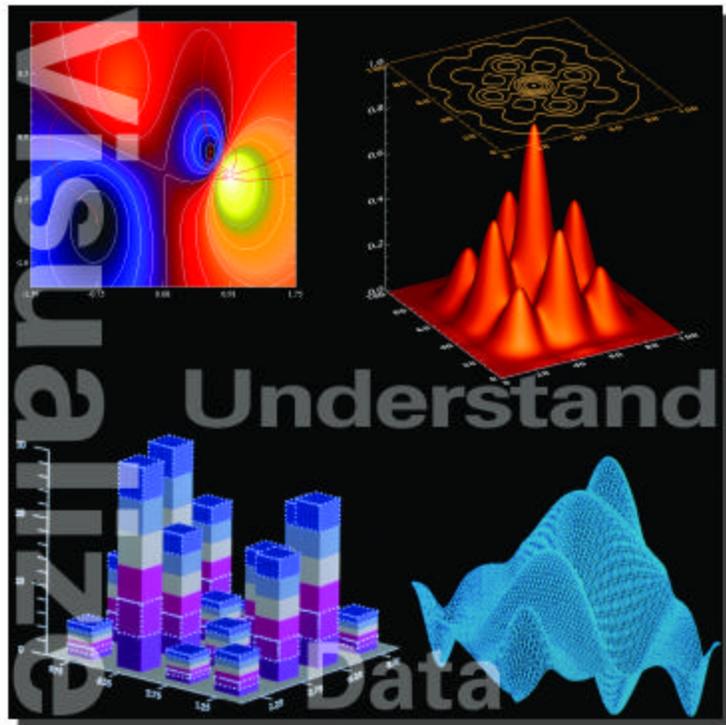




## P V - W A V E 7 . 5<sup>®</sup>



Database Toolkit User's Guide

HELPING CUSTOMERS **SOLVE** COMPLEX PROBLEMS

---

---

# ***Table of Contents***

## ***Chapter 1: Using the Database Toolkit*** 1

Introduction 1

Recommended Directory Structure 2

PV-WAVE Calls to the DB Connection Driver 3

    Error Handling 4

PV-WAVE DB Connection API 4

Code Skeleton 6

Building PV-WAVE with the Driver 8

Testing Tips 9

Conclusion 9



# *Using the Database Toolkit*

---

## *Introduction*

This document is intended for use by application developers, as opposed to PV-WAVE users who want to exercise the Database Link functionality for PV-WAVE. We assume that the reader is familiar with C code, compilers, linkers, Makefiles and other generic tools used by application developers. We also assume that the reader is familiar with programming with database vendors C interfaces, like PRO\*C for Oracle.

This document describes how to write a connection driver for a particular database. Step by step, you will learn how to set up your file structure for development, the API calls necessary to the database connection, the code structure, how to build and link your database connection driver to PV-WAVE kernel. We will also provide a few tips on how to test the connection driver.

VNI supplies

- `dbms_api.h`, an include file that describes a set of constants, error variables used by the connection, and function definitions.
- `template_ddl.c`, a sample code from which the DB connection programmer can use to derive more easily the DB connection code.
- This manual.

The DB Connection developer, to whom this manual is dedicated, writes the DB connection code. The `.c` file will then be compiled and linked with the PV-WAVE libraries to result in a new PV-WAVE executable.

The PV-WAVE user will then be able to call three functions, <dbms>\_sql, <dbms>\_connect, and <dbms>\_disconnect to retrieve and/or write data to/from the DBMS.

---

**NOTE** The **PV-WAVE:Database Connection User's Guide** explains the functions available to PV-WAVE users from the drivers built using this manual.

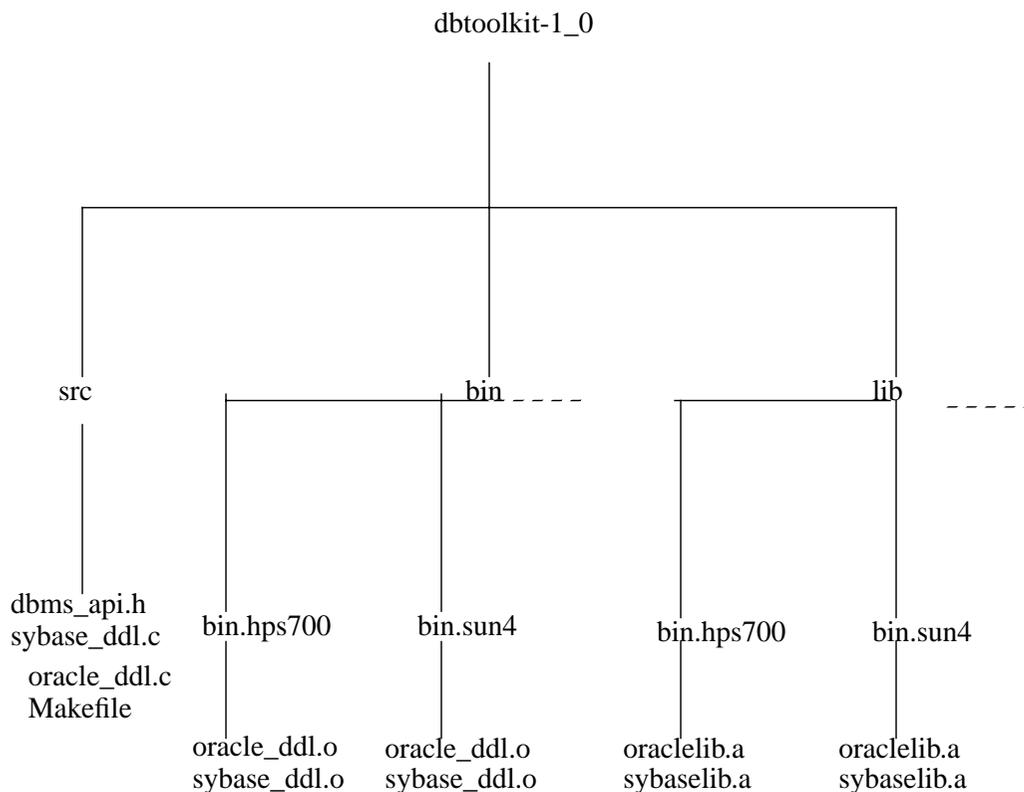
---

---

## ***Recommended Directory Structure***

The developer must be able to develop a number of database connections for a variety of platforms.

We suggest the following file structure to avoid development problems:



dbtoolkits is the top level directory.

The src directory contains the *sybase\_ddl.c*, *oracle\_ddl.c*, one for each driver. The .ddl in these file names mean driver dependent layer, and is a historical naming convention from the design of the database connection). These are the files the DB Connection developer will have to write; to facilitate writing these files, we provide *template\_ddl.c*.

The *dbms\_api.h* provided by VNI contains declarations and definitions of constants, API DB connection external declarations, and error variables that will be needed in the driver code. A *Makefile* can be set so as to be platform dependent, so that building the DB connection driver object files is done according to the operating system specific flags and include files. We advise you to place them under the src directory.

Let us assume you are building two drivers that will run under HP-UX 9.x and SunOS 4.1.x: The bin directory would contain the object code for both drivers, *oracle\_ddl.o* and *sybase\_ddl.o*. These objects files will be needed at link time.

The lib directory will contain the database vendors libraries: *oraclelib.a* could in fact be a series of archive libraries needed to write a C program retrieving data from a database. Oracle, for example, would require the following libraries installed: *libsql.a osntab.o libsqlnet.a libora.a*. When linking the PV-WAVE libraries, you will need to add these libraries to the link command.

Now that you have set up your directory structure, let us look at the hooks to the DB connections and the API calls.

---

## ***PV-WAVE Calls to the DB Connection Driver***

The API calls allow the developer to create a PV-WAVE table and fill each one of the cells in that table; it does not show how the driver itself is called.

This section describes the hooks from which the PV-WAVE commands *db\_connect*, *db\_sql* and *db\_disconnect* are called.

Eleven (11) database names have been chosen, to be called from the Command Line in PV-WAVE: *oracle*, *sybase*, *unify*, *rdb*, *ingres*, *informix*, *dbms1*, *dbms2*, *dbms3*, *dbms4*, *dbms5*. Let us assume that you want to build an INGRES database connection link. When, at the PV-WAVE command line prompt, you type:

```
WAVE> db_handle = db_connect( 'ingres', 'scott/tiger')
```

the *ingres\_connect* function is called with the 'scott/tiger' connection string (this implies that only one connection of type *ingres* can be set at once; however, while the Ingres connection is established, you can ask for an Oracle connection).

When then typing:

```
WAVE> table = db_sql (db_handle, 'select * from emp')
```

the `ingres_sql` function is called with 'select \* from emp' as an argument.

Similarly,

```
WAVE> db_disconnect, db_handle
```

will call the `ingres_disconnect` function.

Of course, the same is valid for the 11 names mentioned above. Writing the DB Link driver consists in writing three functions, corresponding to the three PV-WAVE functions, using the API calls to format the resulting table in PV-WAVE.

## Error Handling

Each one of these functions must return a short, containing either a TRUE for success, FALSE for error; TRUE and FALSE are defined in `dbms_api.h`. The errors must be contained in a string described in `dbms_api.h`: `extern char sql_error_string[]` or `extern char sql_warning_string[]`. At the beginning of each of the hooks, these strings needs to be reset to "\0", so as to avoid repeating the previous errors. The messages passed in these strings will be printed as a PV-WAVE error message, when the return value of the function is FALSE.

The role of the Database connection driver writer is to code a `.c` file, e.g. `oracle_dll.c`, that contains these three hooks, e.g. `oracle_connect`, `oracle_sql` and `oracle_disconnect`, using the API calls provided in the following section, and the database vendor own C interface, like PRO\*C for Oracle.

---

## PV-WAVE DB Connection API

A total of four API calls are needed to ease the transformation of data resulting from an SQL request to a PV-WAVE table variable. Even though you never actually refer to the table in the API calls, a table will be created, and instantiated with these calls. You will see how these calls are used in `template_ddl.c`.

### ***void wave\_struct\_init():***

- input: (long) number of columns.
- output: (void)

- description: the primary space for the PV-WAVE table will be internally reserved. The internal structure for the table will be used by subsequent calls through the API.

***void wave\_define\_dbms\_tag():***

- input: (int) column index, (char\*) column name, (unsigned char) type. The internal table is further defined for each column. The column index must be within 0 (zero) and the number of columns parameter given to the wave\_struct\_init call. The column name is a NULL terminated string, and needs to be allocated/deallocated in the calling driver. The type is one of the values TYP\_BYTE, ..., TYP\_STRUCT given in the dbms\_api.h file.
- output: (void)
- description: the internal PV-WAVE structure for the table is further formed.

***void wave\_make\_sql\_table()***

- input: (int) number of rows for the PV-WAVE table.
- output: (void)
- description: the total internal space for the PV-WAVE structure is finalized. This call needs to be done after the wave\_define\_dbms\_tag call for all of the columns.

***void put\_in\_cell()***

- input: (long) row, (long) column, (char \*) value. The cell location is given by its row and column coordinate. The row and column values are within 0 (zero) and the parameters given respectively to wave\_struct\_init and wave\_make\_sql\_table.
- output: (void)
- description: At the row, column location, given the type of the column specified in the wave\_define\_dbms\_tag call previously, put\_in\_cell will format the string as PV-WAVE data.

---

## Code Skeleton

Now that you are acquainted with the hooks to your driver code from PV-WAVE, and with the API calls that allow you to create and fill a PV-WAVE table, we present a code skeleton that your particular database connection can follow. We consider the case where you want to write a Sybase driver, to avoid confusion. The same is applicable to any other database. You will find the actual skeleton in the `template_ddl.c` file. `Template_ddl.c` will give you a general sense of the possible code for your database connection.

### ***short sybase\_connect( char \* connect\_string)***

- reset the `sql_error_string`, `sql_error_string[0]='\0'`;
- Initialize a Sybase type to PV-WAVE type conversion array, so that `wave_define_dbms_tag`, and `put_in_cell` will be able to create and transform data correctly in `sybase_sql()`.
- check that there is no other sybase connection established (with a static flag for example).
- install Sybase own error and message handling routines
- parse the `connect_string`, and set the Sybase data structure for the Sybase connection; call the Sybase connection routine.
- Check for errors, report them if necessary in `sql_error_string`, and clean up.
- return a TRUE or FALSE status.

This routine does not use any API call to PV-WAVE. It really sets up the structure for the Sybase connection calls (`dbopen`, `dbuse`,...)

### ***short sybase\_sql( char \* command);***

- Reset the `sql_error_string`, and any associated error variable.
- Checks that there is a connection to the database server.
- Make sure that any pending database request has been flushed, especially those that might have caused errors.
- Send the command string to the Sybase server, checking for errors at all step in the process.
- Get the number of columns returned from the select command, and call `wave_struct_init`. Make sure there is at least one column returned.

- Get each column name, and the Sybase type. For each column, call `wave_define_dbms_tag` with the proper parameter, using the conversion array initialized in `sybase_connect` to provide the correct PV-WAVE type to the column.
- Get the number of rows returned from the SQL select query. Make sure it is at least one. Call `wave_make_sql_table` with number of rows as its argument. You now have set up the table internal data structure and reserved the necessary space for it.
- For each cell returned, call `put_in_cell` with the row, column and value as a string. until no more cell is available. ( A cell is an element, addressable via a row and a column).
- Check for any error either from your code, or from Sybase calls, and set `sql_error_string` accordingly.
- Return either TRUE or FALSE. If FALSE is returned, the `sql_error_string` string will be shown from PV-WAVE.

---

**NOTE** The `db_sql` function has been set up for importing data into PV-WAVE. However, exporting data to the database can be done, since you can pass any SQL strings to `db_sql`. When the SQL command is not a query, you will need to set up a pseudo-table: we suggest you create a one column, one row table, and pass a TRUE FALSE value into the single cell. That way, you will be able to assess the status of the call.

---

Based on the command string, you can then either create the status pseudo-table or a real table filled with data from the database server. Your PV-WAVE application needs to manage the strings passed to the SQL command of `db_sql`. PV-WAVE variables do not possess any attribute that bind them to actual database table; hence the PV-WAVE application will build strings from PV-WAVE variables and tables to update the corresponding values in the database.

A PV-WAVE call could look like:

```
WAVE> status = db_sql(db_handle, $
    "insert into dept values (50, 'PRODUCTION', 'SAN FRANCISCO')")
```

The PV-WAVE programmer will have build the SQL command string “insert into dept values (50, ‘PRODUCTION’, ‘SAN FRANSICO’)” from PV-WAVE variables.

### ***short sybase\_disconnect()***

- Check that there is a connection
- reset the error string
- clean up your data structures
- Disconnect the Sybase connection (in Sybase, dbexit())
- Check for errors, set the error string
- Return TRUE or FALSE.

This skeleton is only given as an example of how to write the code. For example, you could write a conversion function from Sybase to PV-WAVE types, or have a static array predefined.

It is necessary that you verify the values passed to the PV-WAVE API calls, and that these calls are in sequence as each one rely on the previous for the correct information.

---

## ***Building PV-WAVE with the Driver***

For each one of the driver, an object file will be generated for a platform. For example, a database connection driver for Oracle on HP 9000 series 700 object file will be created as `oracle_ddl.o` under the `dbtoolkit-1_0/bin/bin.hp700` directory.

To link this object file with PV-WAVE, you will need to modify `quick.mk`, or your own linking command, to make sure that `oracle_ddl.o` is linked in the proper order:

The link command must have the following order:

- 1: wave libraries except `dbms.hps700.a` and `wave.hps700.a`
- 2: your driver or drivers, for example `oracle_ddl.o`, `sybase_ddl.o`
- 3: `dbms.hps700.a`
- 4: the database vendor's libraries, for example the `hps700 libora.a`, `libsqlnet.a`, `libsql.a`, `onstab.o`
- 5: `wave.hps700.a`
- 6: the other required libraries, `Xm`, `Xt`, `X11` and so on.

The link command will indicate that there some symbols are defined more than once. That is because the database driver that you have written contains and over-

rides the same symbols that are used in `dbms.hps700.a`; in `dbms.hps700.a` are some stub functions for the ten database hooks. For example, `oracle_connect` belonging to the oracle hooks. As you can see, you can link in up to ten drivers with PV-WAVE code, each based on some particular vendor's library.

---

## ***Testing Tips***

This section suggests some particular areas that the database connection driver programmer may want to test in depth. We assume in this section that you have just built an Oracle connection on HP.

Each database vendor has its way to access databases. To test the `db_connect` routine, make sure that you have access to SQLPLUS. The connection string you parse in `oracle_connect` should be able to specify a remote system accessible from SQLPLUS.

Test the connection with a user that does not exist to make sure that the error string is set properly from the driver.

Connect twice in a row to make sure you are not allowing this.

Connect, disconnect, and reconnect to the server.

Retrieving data from the database needs to be tested for each type of data: we suggest you create a table containing all the types from supported by the Oracle, and try to retrieve that table in its entirety. This will test that the conversion you built from Oracle types to PV-WAVE types is correct.

Create a table that contains only one row and one column, to test the boundary, and retrieve the table for boundary testing.

Create a table that does not contain any row to make sure you are testing that case.

Query a table that does not exist to make sure the error string is set properly.

---

## ***Conclusion***

You now have created a database connection for PV-WAVE. Let us pretend it was a Sybase link. PV-WAVE users are now able to call

```
sybase=DB_CONNECT('Sybase', '<connection_string>'),
```

which in turns calls your `sybase_connect` code to initiate a connection.

PV-WAVE users are also able to call

```
TABLE=DB_SQL(sybase, '<select_string>')
```

which calls your `sybase_sql` code, which in turns calls the API functions to set up the `PV-WAVE` variable returned to the user.

Similarly, `PV-WAVE` users can now disconnect from the Sybase DBMS by calling

```
DB_DISCONNECT, sybase
```

which will call your `sybase_disconnect` function.

`DB_CONNECT`, `DB_DISCONNECT` and `DB_SQL` are all described in more detail in the *PV-WAVE:Database Connection User's Guide*.