SYBASE

Fundamentals of DB-Library Student Guide

™DB-Library Course Topics

SYBASE

DB-Library Overview

Objectives

- Understand the function of DB–Library and its relation to the SQL Server and application programs

- Familiarity with the components of DB–Library and the location of these components

- Learn the steps required to build a "runnable" application

Function of DB-Library

- **Main function: manages communication and data transfer between the SQL Server and the applications**
  1. Transmit SQL statements to the SQL Server
  2. Return data to the program

- **Provides a consistent programming interface across multiple languages:**

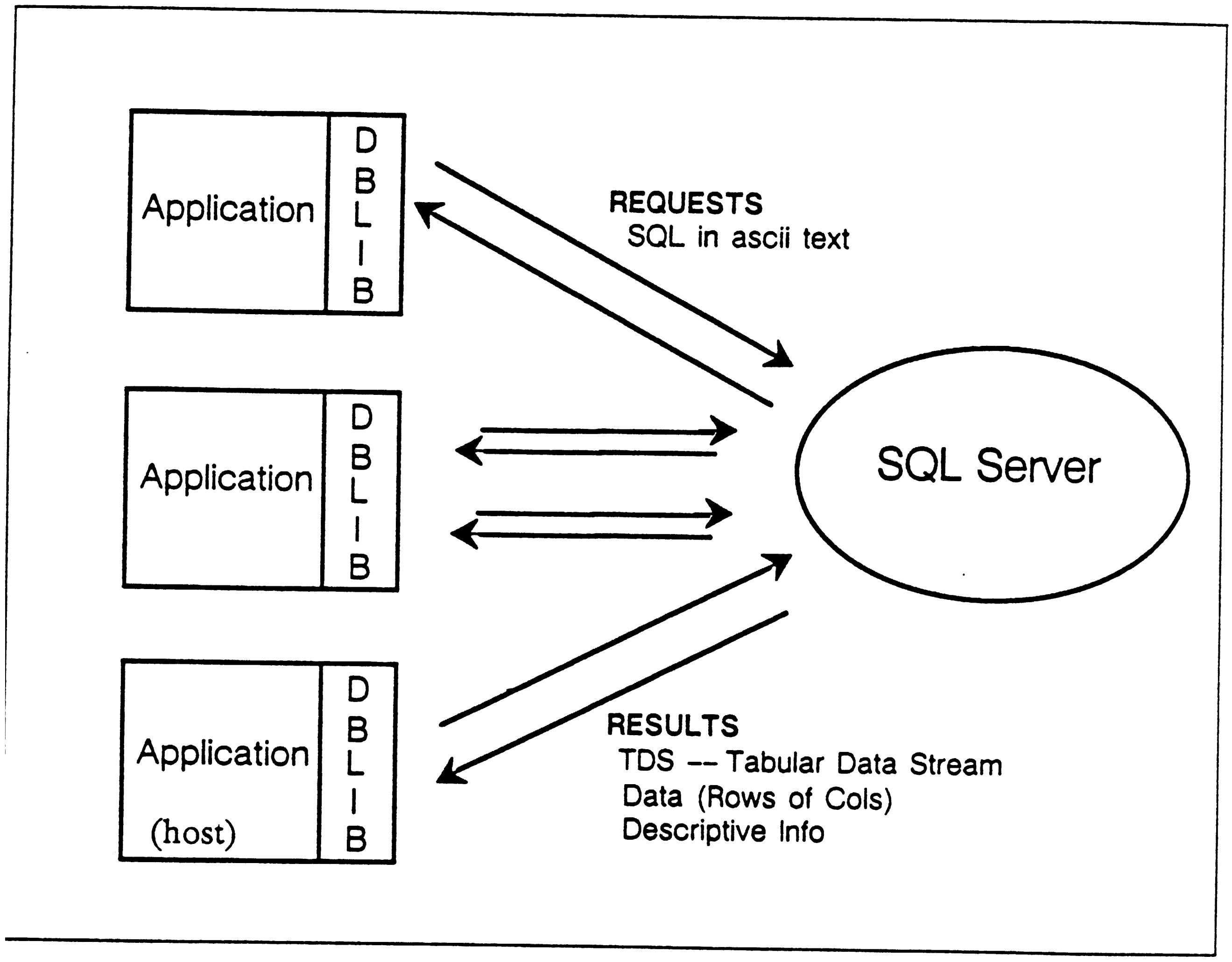
  Supported Languages:
  C
  Fortran
  Cobol
  Ada
  Pascal

- **Provides subroutine/function calls**

- **Provides standard structure and type definitions**

- **Insulates application programs from the changes to SQL Server internals**

DB-Library and the SQL Server



- Requests to the SQL Server are in SQL text
- Results to the application are in **host** datatypes
- SQL Server uses an application protocol called 'TDS' (Tabular Data Stream)

What does DB-Library provide?

- **Initialization and Clean-Up functions**

  Establish connection to SQL Server

  Close connections

  Login and use databases

- **Command Set-up and Execution**

  Build SQL command batches

  Send commands to SQL Server

- **Results Processing**

  Majority of functions fall in this category

  Process results singly or a buffer at a time

- **Miscellaneous Routines**

  Get information about the data

  Data conversion routines

  Query status of a process or command

Conventions used by DB-Library

- **Function Names**

  All functions are dbxxx (for C) or fdbxxx (for Fortran) and are typically lower case

  In C some macros are defined with names in uppercase, but their usage is similar to functions

- **Parameters**

  Most functions require some parameters

  Some functions accept NULL as a parameter

- **Returns from functions**

  Many functions will return a value which can be checked by the program

  In Fortran, functions can be called either as functions (when you want to check the value) or as subroutines (when the return value is ignored)

  ie., call fdbname (x,y) or return = fdbname(x,y)

  Most function returns are defined symbolically for you

- **All functions, parameters and returns are fully documented in the DB-Library reference manuals for the appropriate language**

What else does DB–Library provide?

- **Return Value Definitions such as:**

RETCODE	NO_MORE_ROWS
FALSE	MORE_ROWS
TRUE	BUF_FULL
SUCCEED	NO_MORE_RESULTS
FAIL	REG_ROW
NULL	

- **Return Codes such as:**

INT_EXIT	DBSAVE
INT_CONTINUE	DBNOSAVE
INT_CANCEL	DBNOERROR

- **Structure Definitions such as:**

DBPROCESS	LOGINREC

- **Type Definitions such as:**

DBINT	POINTER	DBMONEY

DB-Library Files (Unix - C)

- **$SYBASE/include**

  Definitions are contained in header files:

sybfront.h	must be included first contains type definitions
sybdb.h	defines structures;
sybdbtokens.h sybloginrec.h	included automatically by sybdb
syberror.h	contains error severity definitions

- **$SYBASE/lib**

libsybdb.a	Contains the code for all the functions and macros

- **Usage**

  In your C program, begin the program with:

```
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
```

  Specify the library file when linking the program

DB–Library Files (VMS – Fortran)

- **SYBASE$SYSTEM:[SYBASE.INCLUDE]**

  Contains a header file with definitions of parameter and function return values, to be included with the Fortran program.

  All the appropriate C files are converted and combined into one text library file

  File name: FSYBINC.TLB

- **SYBASE$SYSTEM:[SYBASE.LIB]**

  Fortran programs require two Sybase link libraries, in addition to the standard system libraries:

LIBFSYBDB.OLB	provides the interface to the C library
LIBSYBDB.OLB	identical to the C library on Unix

  Libraries can be linked shareable or non–shareable

- **Usage**

  In your Fortran program, begin the program with:

  ```
  include '(fsybdb)'
  ```

  Specify the libraries when linking the program

Compiling & Loading

(Unix - C)

- Define **SYBASE** if necessary

  ```
  setenv SYBASE /usr/u/sybase/...
  ```

- For compilation, add the include files from **$SYBASE/include**

  ```
  cc myprogram.c -I$SYBASE/include
  ```

- For loading, add the library files from **$SYBASE/lib**

  ```
  cc myprogram.c
      -I$SYBASE/include
        $SYBASE/lib/libsybdb.a -o output
  ```

- For efficiency, use a **make** file

  A sample makefile is in the appendix

Compiling & Loading

(Fortran)

- For compilation: (add to **LOGIN.COM**)

```
$DEFINE FORT$LIBRARY
   SYBASE$SYSTEM:[SYBASE.INCLUDE]FSYBINC.TLB
FOR myprog.for /warn = dec
```

- For linking, sharable make a **LINK.COM file**

```
$ LINK myprog, -
   SYBASE$SYTEM:[SYBASE.LIB]LIBFSYBDB/LIB,-
  SYBDB_OPTIONS/OPT, SYS$INPUT/OPT
    SYS$LIBRARY:VAXCRTL/SHARE
```

- For linking, non-sharable put defines in **LOGIN.COM** file

```
$ DEFINE SYS$LIBRARY SYS$SYSROOT:[SYSLIB]
$ DEFINE LNK$LIBRARY
      SYBASE$SYSTEM:[SYBASE.LIB]LIBFSYBDB.OLB
$ DEFINE LNK$LIBRARY_1
      SYBASE$SYSTEM:[SYBASE.LIB]LIBSYBDB.OLB
$ DEFINE LNK$LIBRARY_2 SYS$LIBRARY:VAXCRTL.OLB
LINK myprog
```

Summary

- **DB–Library is a software package containing:**

  Definition files which you include in your source;
  Library files which you link with your program.

- **DB–Library components**

  Header/include files:

C	Fortran
sybfront.h	fsybinc.tlb
sybdb.h	
syberror.h	
sybtokens.h	
sybloginrec.h	

  Library files:

C	Fortran
libsybdb.a	libfsybdb.olb
	libsybdb.olb

Lab Exercise: Environment Set-Up.

Lab Time: 20 minutes

The purpose of this lab is to have you set up your user account properly for the remainder of the course.

All labs assume you have logged in to the operating system as userN (where N is indicated on your terminal) for Unix, and USERN for VMS. The password is the same as the login name.

1. Unix/C: copy **/usr/u/train/dblib/makefile** into your home directory.

   VMS/Fortran: Copy **SYBASE$SYSTEM:[SYBASE.TRAIN]DBLINK.COM** into your home directory.

2. Create a program in your selected language (C or Fortran) which simply prints your name on the terminal. The program should begin with all the necessary DB-Library include statements. To verify that all the linkage is done correctly, we will add one call to the library: put a call to dbexit at the end of the program. (In Fortran, use call fdbexit( ). In C, use dbexit( )).

3. Modify the LINK.COM or makefile to reference your program.

4. Compile, link and run your test program.

   Unix/C:

   1.) To compile and link: **make lab**
   2.) To run: **lab**

   VMS/Fortran:

   1.) To compile: **for/warn=dec <your program>**
   2.) To link: **@DBLINK.COM**
   3.) To run: **run <your program>**

Lab Answer

```
/* Lab Number 1 */
/* This program simply sets up a program to use DB-Library procedures */
/* and then prints out a name. Makes one DB-Library call to check */
/* that the linking was done properly */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

main()
{
        printf("Hello, this is the Sybase example for Lab 1\n");

        /* make a DB-Library call */
        dbexit( );
}
```

SYBASE

Module 2
Getting Started

Objectives

- Understand communication between applications and the SQL Server

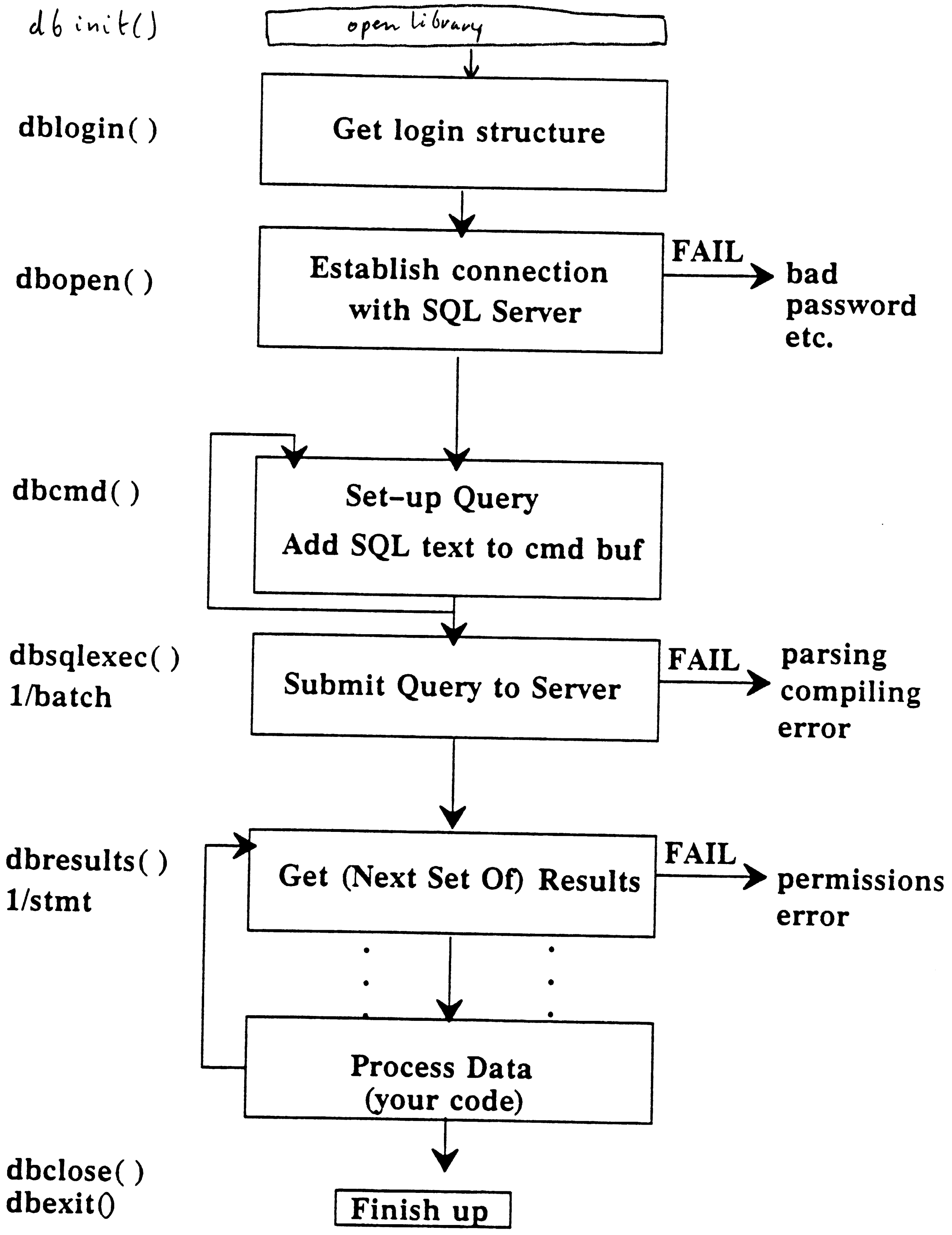
- Establish a connection to any server as any user

- Construct and send SQL statements to the server

- Display the results returned

- Properly terminate applications programs

Application Overview



Simple Program

```
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
main()
{
    DBPROCESS               *dbproc;
    LOGINREC                *login;
    RETCODE                 return_code;
    if (dbinit() == FAIL) exit(ERREXIT);
    login = dblogin( );
                                        <----- DBSETLUSER(login, "user5");
    dbproc = dbopen(login, NULL);              DBSETLPWD (login, "user5");
    if (dbproc == NULL)                        DBSETLAPP (login, "demo5");
            exit(ERREXIT);

    dbcmd (dbproc, "select * from publishers");
    if (dbsqlexec(dbproc) == FAIL)
    {
            dbexit( );
            exit(ERREXIT);
    }

    while ( dbresults(dbproc) != NO_MORE_RESULTS )
            dbprrow (dbproc);

    dbexit ( );
    exit(STDEXIT);
}
```

Fortran Simple Program

```
      program Simple
      include '(fsybdb)'
      INTEGER*4     dbproc
      INTEGER*4     login
      login = fdblogin()
      dbproc = fdbopen(login,NULL)
      if (dbproc .eq. NULL) then
               call exit
      end if

      call fdbcmd(dbproc,' select * from publishers')

      if ( fdbsqlexec(dbproc) .eq. FAIL) then
               call fdbexit( )
               call exit
      end if

      do while ( fdbresults(dbproc) .ne.
     2                   NO_MORE_RESULTS)
          call fdbprrow(dbproc)
      end do

      call fdbexit ( )
      call exit
      END
```

Login Record

- **Function**

  Data structure which is used to describe database user in order to establish connections to the SQL Server

  Allows the program to set values for user name, password and other parameters

- **Usage**

  1. Declare a pointer (handle) to a LOGINREC structure

C	FORTRAN
LOGINREC *login	Integer*4 login

  2. Call DB–Library to allocate and initialize the structure

login = dblogin()	login = fdblogin()

Modifying the Login Record

- Changing the defaults

C	FORTRAN

DBSETLUSER(login,"user")
DBSETLPWD(login,"pass")
	call fdbsetluser(....)
	call fdbsetlpwd(....)

Note: to declare a null password, say
DBSETLPWD(login, "") in C.
fdbsetlpwd(login, NULL) in Fortran

- **Examples of when you would want to do these calls**

In VMS, when the login name is in upper case, and SQL Server login name is in lower case .

When reading a password from the user

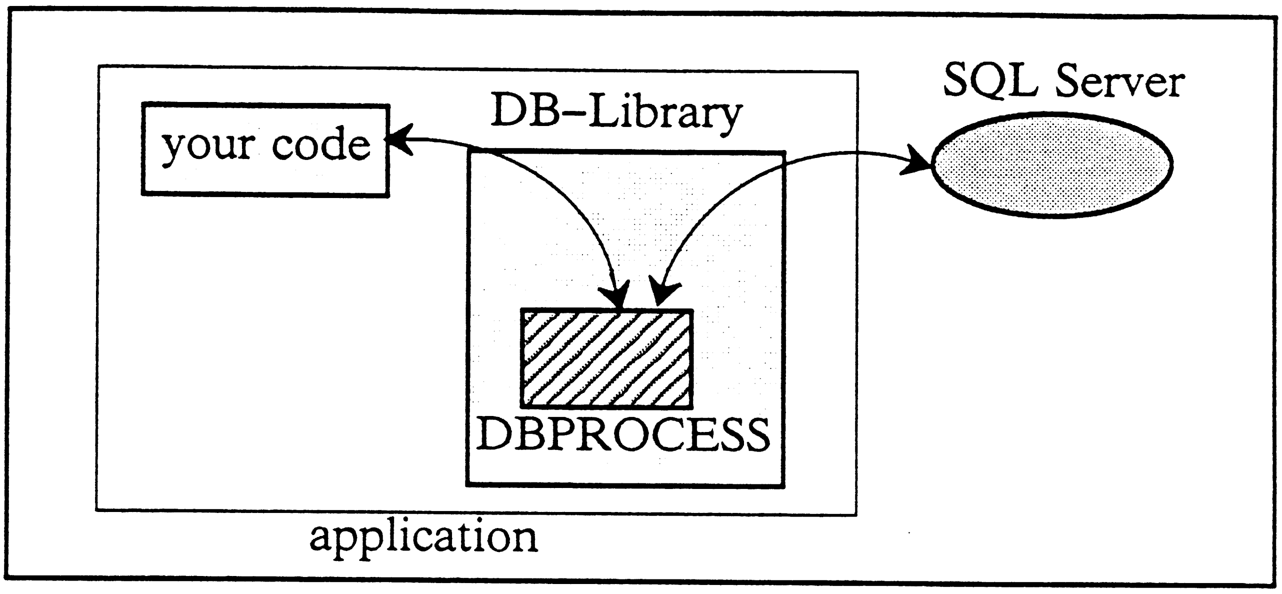
When you want to establish a connection to the SQL Server with a general or group login rather than the user login.

- **Examples of when you can omit these calls**

When the login name is the user's operating system name

When the user's password is a NULL password

DBPROCESS



- **Function**

  Provides the basic data structure for communication between the application program and the SQL Server

  Typical contents include:
  SQL Command buffer
  Data which was returned from the server
  Meta Data (information about SQL results)
  Status information

  It is the first argument in almost every DB–Library call

dbopen( )

- **Function**

  Pass the information in the LOGINREC to the specified SQL Server

  If the login is successful, allocate space for the DBPROCESS structure and initialize its fields

  Optional parameter: the server name to connect to, ie.,
  dbproc = dbopen (login, "servername")

- **Returns:**

  Pointer to the DBPROCESS structure if successful;

  NULL if any error occurs

- **Usage**

C

```
LOGINREC        *login;
DBPROCESS       *dbproc;
login = dblogin( );
dbproc = dbopen (login, NULL);
      if (dbproc == NULL)
            /* error, exit. */
```

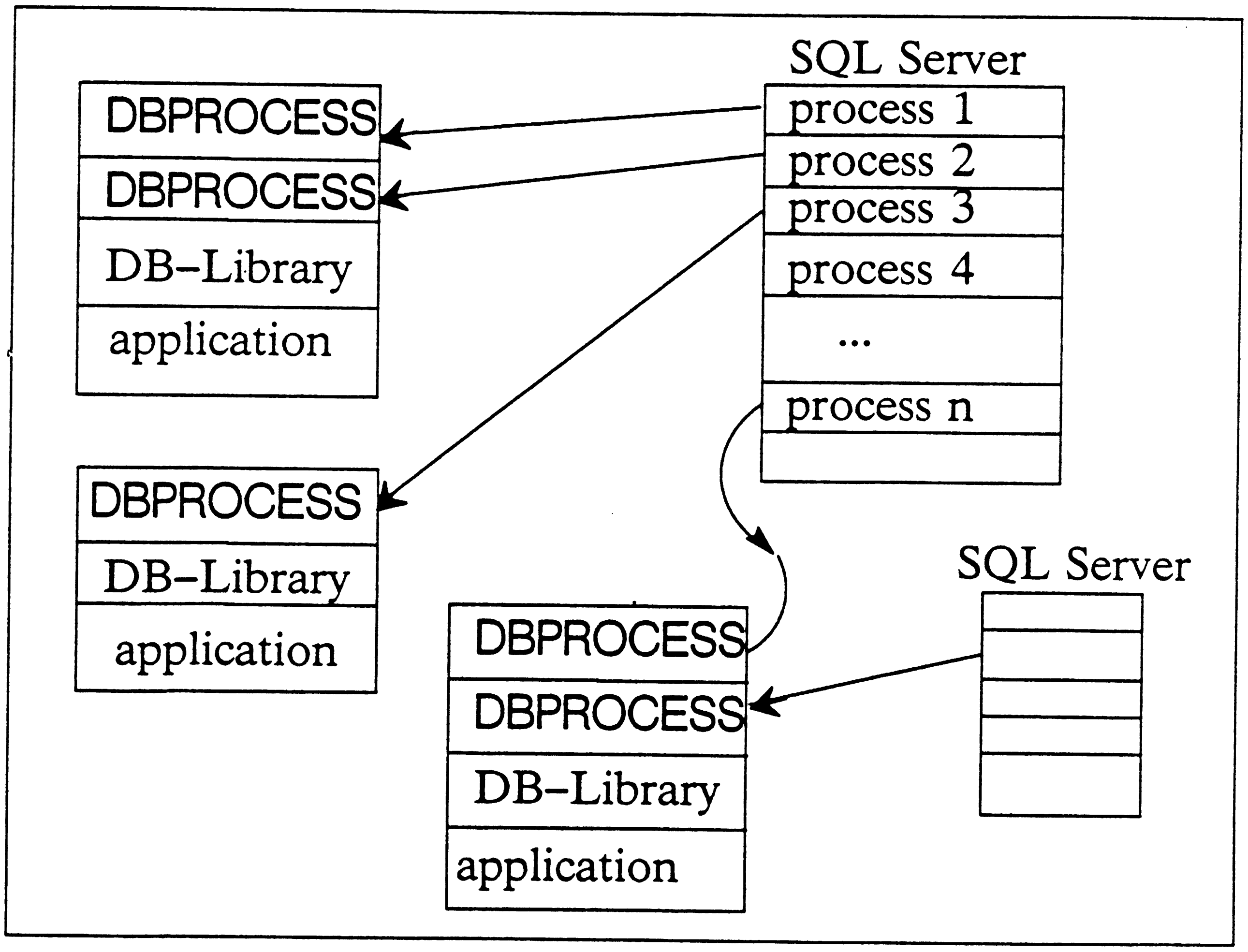
Fortran

```
INTEGER*4   login
INTEGER*4   dbproc
login = fdblogin
dbproc = fdbopen(login, NULL)
if (dbproc .eq. NULL) then
      /* error, exit */
end if
```

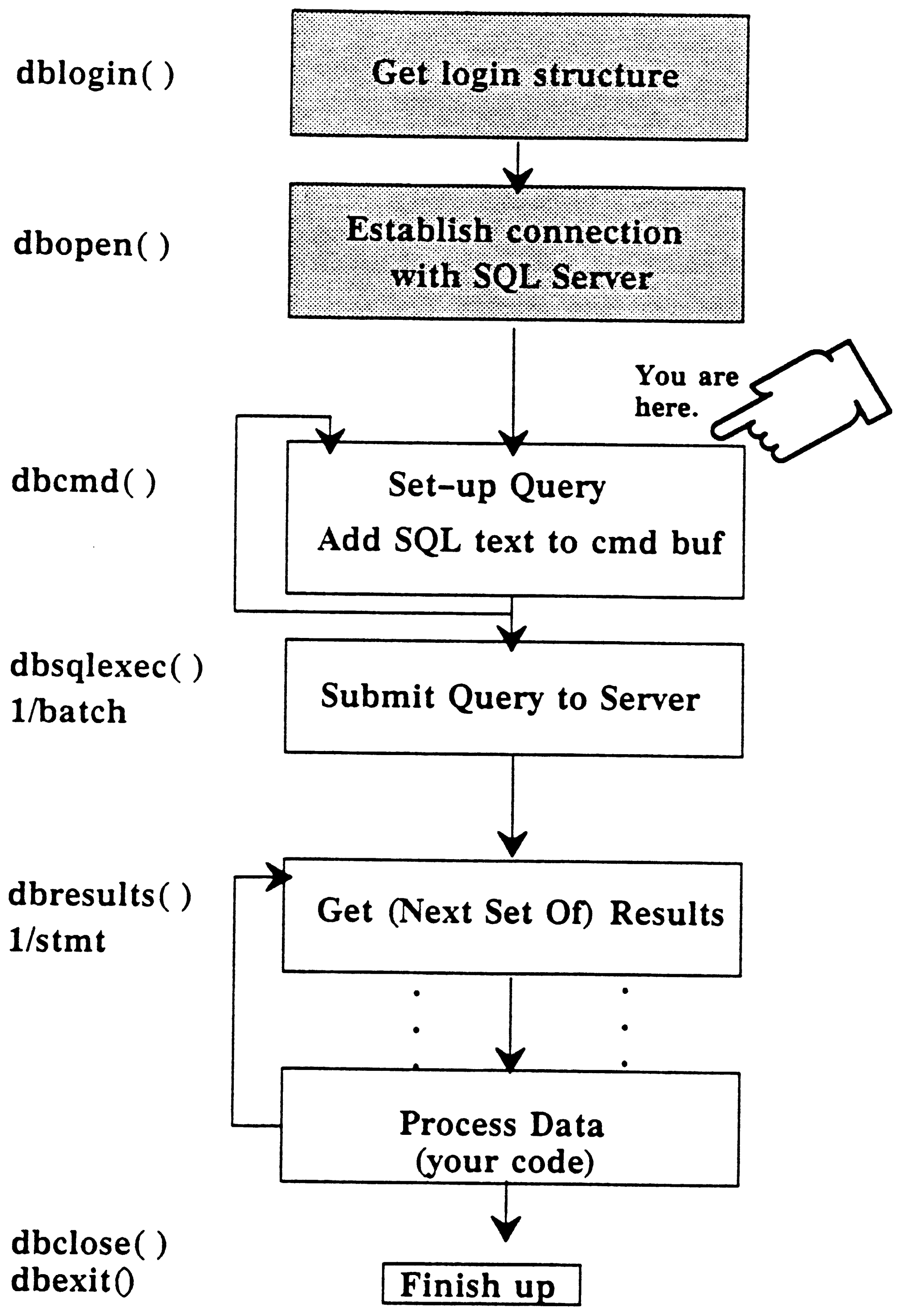
- **Errors can be caused by bad login information, server not running, wrong servername, etc. Do Not proceed using DB-Library calls with a NULL dbproc.**

DBPROCESSes and the SQL Server

- For each separate DBPROCESS in an application, there is a unique server process on the SQL Server
- An application can open multiple DBPROCESSes to multiple servers, and/or multiple DBPROCESSes to one server



Where are we now?



Building SQL Commands

- **Syntax**

  dbcmd (dbproc, "SQL - ascii text"); or
  dbcmd (dbproc, Pointer to a string); or

- **Command batches**

  One or more SQL commands are accumulated in a buffer (in the DBPROCESS structure) using DB–Library calls

  A batch of commands provides efficient access to the SQL Server since all the commands are sent across the network at one time

  All commands in the batch must parse correctly in order for any results to be returned

  Each command is literally appended to the buffer; it is the program's responsibility to provide separators (blanks are sufficient) between commands.

dbcmd

- **C Examples**

```
1.   dbcmd (dbproc, " select * from authors");
     dbcmd (dbproc, " where au_id > 10");
                     ↑

2.   dbcmd (dbproc, " sp_help authors");
     dbcmd (dbproc, "   execute sp_help sales");
                      ↑ spatie
3.   char  *sqltext;
     .....
     sqltext = " select * from pubs..sales";
     dbcmd (dbproc, sqltext);
```

- **Fortran example**

```
     CHARACTER *(28)    sqltext
     ...
     sqltext = ' select * from pubs.dbo.sales'
     call  fdbcmd (dbproc, sqltext)
     call  fdbcmd (dbproc, '  where qty > 50')
```

- **Notes:**

Trailing blanks are truncated in Fortran; separating blanks should be put at the beginning of the string

The command buffer is cleared by the next **dbcmd** after the buffer has been sent to the SQL Server

Sending Commands

- **When you have all the SQL put together . . .**

  return_code = dbsqlexec (dbproc);

  - The command buffer is sent to the SQL Server and then DB–Library waits for the first results

  - Entire batch must be syntactically and semantically correct or routine will fail (determined by SQL Server)

- **Returns**

  SUCCEED – guarantees at least one valid set of results

  FAIL – Syntax, illegal objects, etc.

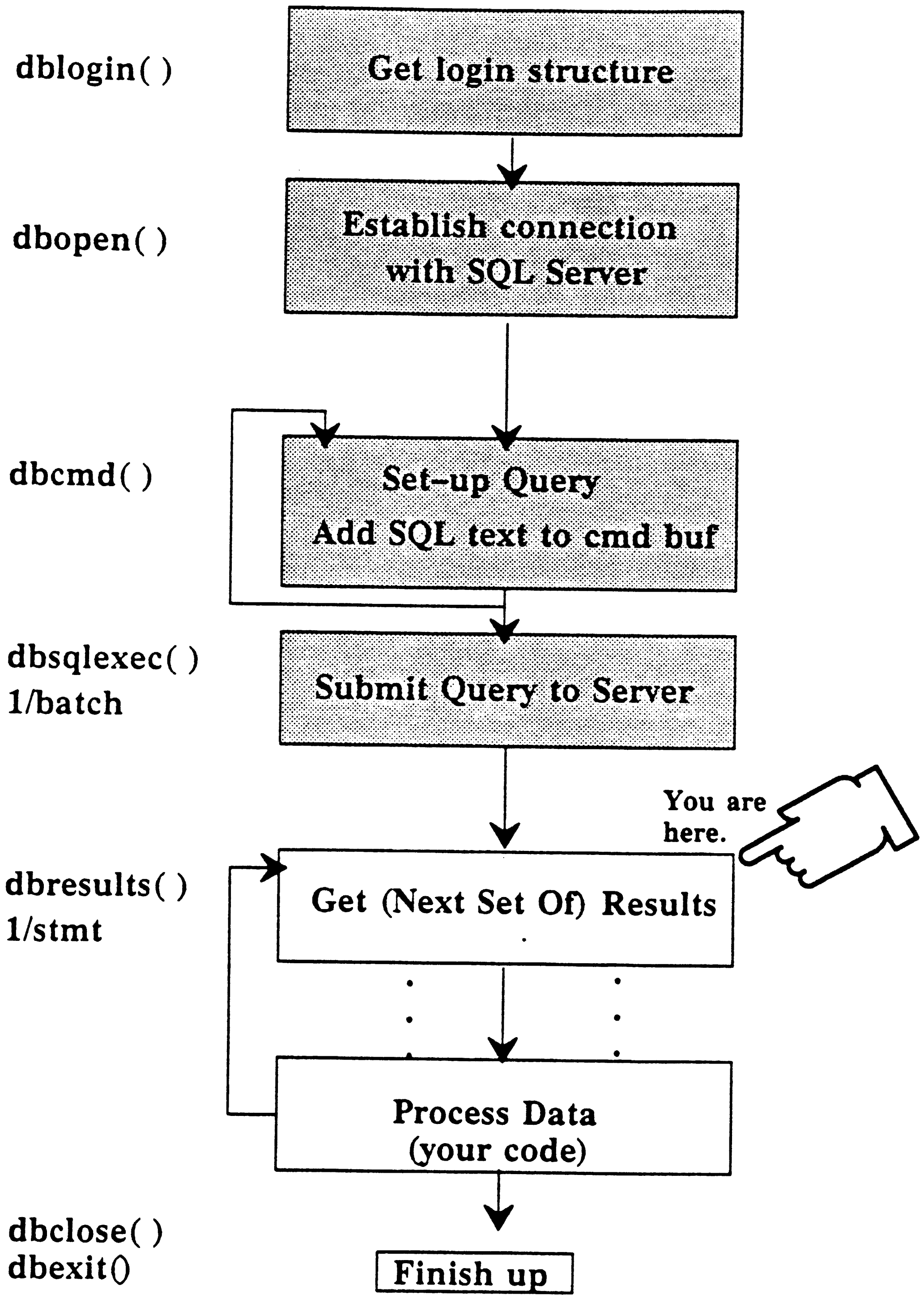
  FAIL will not occur for run–time errors such as protection violation unless it is the only command in the buffer

- **Typical code**

```
C:  if (dbsqlexec(dbproc) == FAIL)
        /* exit or error processing */

Fortran:  if (fdbsqlexec(dbproc) .eq. FAIL) then
                exit....
```

Now What?


dblogin( )
Get login structure
dbopen( )
Establish connection with SQL Server
dbcmd( )
Set-up Query
Add SQL text to cmd buf
dbsqlexec( )
1/batch
Submit Query to Server
You are here.
dbresults( )
1/stmt
Get (Next Set Of) Results
Process Data
(your code)
dbclose( )
dbexit()
Finish up

dbresults

- **Call dbresults for each command in the batch**
- **Returns**

  SUCCEED – there are results to process

  FAIL – a run time error, such as protection

  NO_MORE_RESULTS – useful when multiple commands have been sent in one batch
- **Typical Usage**

  C:

```
while (dbresults(dbproc) != NO_MORE_RESULTS)
     /* do something */ test one up SUCCEED of FAIL
```

  Fortran:

```
do while ( fdbresults(dbproc) .ne. NO_MORE_RESULTS)
     ...
end do
```

- **What to do with results?**

  In the next module we will discuss **dbnextrow** and **dbbind** to access data

  Normally, reference the data, or move the data to internal variables for formatting and processing;

  For quick access use **dbprrow** to dump all the results to the screen in a pre-formatted style.

Finishing Up

- **dbclose (dbproc)** sluit 1 connectie

  Cleans up the DBPROCESS and deallocates the space

  Terminates the matching process on the SQL Server

- **dbexit ( )**

  sluit alle connecties.

  Terminates all DBPROCESSes currently open for this application.

  dbexit does <u>not</u> exit the program

Putting it all together - C

- **Set up login and connect to server**

```
DBPROCESS          *dbproc;
LOGINREC           *login;
RETCODE            return_code;
login = dblogin( );
dbproc = dbopen(login, NULL);
if (dbproc == NULL)
        exit(ERREXIT);
```

- **Build and send command**

```
dbcmd (dbproc, "select * from publishers");
if (dbsqlexec(dbproc) == FAIL)
{
        dbexit( );
        exit(ERREXIT);
}
```

- **Ready results, and dump to screen**

```
while ( dbresults(dbproc) != NO_MORE_RESULTS )
                dbprrow (dbproc);
```

- **Finish up**

```
dbexit( );
exit(STDEXIT);
```

Putting it all together - Fortran

- **Set up login and connect to server**

```
program Myprog
include '(fsybdb)'
INTEGER*4          dbproc
INTEGER*4          login
login = fdblogin( )
dbproc = fdbopen(login,NULL)
if (dbproc .eq. NULL) then
             call exit
end if
```

- **Build and send command**

```
call fdbcmd(dbproc,' select * from publishers')
if (fdbsqlexec(dbproc) .eq. FAIL) then
             call fdbexit( )
             call exit
end if
```

- **Ready results, and dump to screen**

```
do while (fdbresults(dbproc) .ne. NO_MORE_RESULTS)
       call fdbprrow(dbproc)
end do
```

- **Finish up**

```
call fdbexit ( )
call exit
END
```

Lab Exercise: Accessing the Server and getting results.

Lab Time: 45 minutes

All programs should include proper termination procedures for closing SQL Server connections.

1. Log in to the operating system as userN (where N is a number from 1 to 15 as indicated on your terminal).

2. Write a program which connects to the SQL Server using the defaults (your operating system name, and a null password). The program should run the stored procedure **sp_who** and display the results using **dbprrow.**

3. Write a program to connect to the SQL Server as userN, and send a batch of SQL to the SQL Server. The batch should contain at least two commands, such as `select * from sales`, followed by `select * from authors`. Display the results of all the queries on the terminal, using **dbprrow**.

Optional:

4. Modify the program to connect to the SQL Server as the user named "clerkN", password "clerkN" (where N is the number on your terminal, ie, 1 to 15.)

Optional:

5. Write a program to establish two connections to the SQL Server. One connection is logged in as userN, the other connection is logged in as clerkN. Use one of the connections to run the stored procedure **sp_who** and display results to the terminal. You should see that you have two processes running on the server.

Lab Answers

Problem 2

```
/* Lab Number 2.2 */
/* This lab connects to the DataServer using the defaults, then */
/* runs the stored procedure sp_who */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

main( )
{
        DBPROCESS     *dbproc;
        LOGINREC      *login;
        RETCODE       return_code;

        login = dblogin();

        dbproc = dbopen(login,NULL);
        if (dbproc == NULL)
        {
                printf("No  dbproc\n");
                exit(ERREXIT);
        }

        dbcmd(dbproc, "sp_who");
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( (return_code = dbresults(dbproc)) != NO_MORE_RESULTS)
        {
                if (return_code != FAIL)
                        dbprrow(dbproc);
        }

        dbexit( );
        exit(STDEXIT);
```

Problem 3

```
/* Lab Number 2.3 */
/* This lab connects to the DataServer using the defaults, then */
/* runs a batch of SQL commands, at least two selects */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

main( )
{
        DBPROCESS    *dbproc;
        LOGINREC     *login;
        RETCODE      return_code;

        login = dblogin( );

        dbproc = dbopen(login,NULL);
        if (dbproc == NULL)
        {
                printf("No  dbproc\n");
                exit(ERREXIT);
        }

        dbcmd(dbproc, "select * from publishers");
        dbcmd(dbproc, " select * from sales");

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( (return_code = dbresults(dbproc)) != NO_MORE_RESULTS)
        {
                if (return_code != FAIL)
                        dbprrow(dbproc);
        }

        dbexit( );
        exit(STDEXIT)
}
```

Problem 4

```
/* Lab Number 2.4 */
/* This lab connects to the DataServer,but this time as clerkN, */
/* password clerkN */
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

main( )
{
        DBPROCESS    *dbproc;
        LOGINREC     *login;
        RETCODE      return_code;

        login = dblogin( );

        /* change "clerkN" to specific clerk number when testing */

        DBSETLUSER(login, "clerkN");
        DBSETLPWD(login, "clerkN");

        dbproc = dbopen(login,NULL);
        if (dbproc == NULL)
        {
                printf("No  dbproc\n");
                exit(ERREXIT);
        }

        dbcmd(dbproc, "select * from publishers");
        dbcmd(dbproc, " select * from sales");

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( (return_code = dbresults(dbproc)) != NO_MORE_RESULTS)
        {
                if (return_code != FAIL)
                        dbprrow(dbproc);
        }

        dbexit( );
        exit(STDEXIT)
}
```

Problem 5

```
/* Lab Number 2.5 */
/* This lab opens two separate DBPROCESSes with the DataServer, once */
/* as the default, and once as 'clerkN', password 'clerkN', and runs */
/* the stored procedure 'sp_who' with one of them.  Notice when this */
/* is run that both DBPROCESSes are listed, with different login names */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

main( )
{
        DBPROCESS     *dbproc1;
        DBPROCESS     *dbproc2;
        LOGINREC      *login;
        RETCODE       return_code;

        login = dblogin( );

        dbproc1 = dbopen(login,NULL);

        /* now change users, start another dbprocess */
        DBSETLUSER(login, "clerkN");
        DBSETLPWD(login, "clerkN");
        dbproc2 = dbopen(login,NULL);

        if (dbproc1 == NULL || dbproc2 == NULL)
        {
                printf("Failed dbprocess\n");
                exit(ERREXIT);
        }

        /* now, execute sp_who with dbproc1. */
        dbcmd(dbproc1, " exec sp_who");

        if (dbsqlexec(dbproc1)== FAIL)
        {
                printf("dbsqlexec for dbproc1 failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( (return_code = dbresults(dbproc1)) != NO_MORE_RESULTS)
        {
                if (return_code != FAIL)
                        dbprrow(dbproc1);
        }

        dbexit( );
        exit(STDEXIT);
}
```

SYBASE

Module 3

Error Handling & SQL Parameters

Objectives

- Set up handling for SQL Server messages and errors
- Set up handling for DB–Library errors
- Send SQL text with embedded parameters
- Change the database context

Error Handling

- **Errors and/or messages may be generated by:**

  SQL Server

  DB–Library

  Operating System

- **May be informational or error indications**

  SQL print statements generate information messages

  SQL syntax errors, protection problems, etc., generate error messages

- **Errors and messages can be handled two ways:**

  Programs can check, retrieve and display messages as part of the main program logic

  Programs can be "interrupted" asynchronously whenever a message or error occurs (using error handlers)

- **Error handlers are the preferred way**

  Provide central, standard error and message handling

  Easy to set up

  Programmers can supply separate handlers for
    SQL Server messages and errors
    DB–Library errors

How Error Handlers Work

- **Programmer's responsibility**

  Write the error handler

  Tell DB-Library to use it (dbmsghandle, dberrhandle)

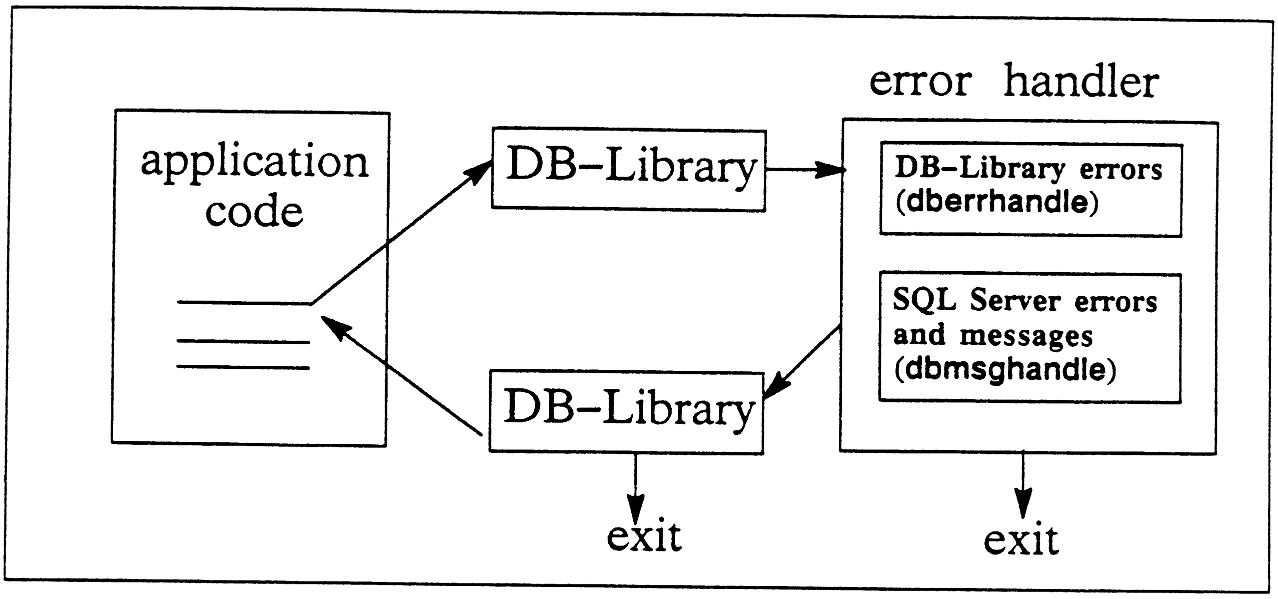
  Test for Failure conditions in the main line code as necessary

- **DB-Library actions**

  Transfer control to the handler when any error/message occurs

  Return to the main program or exit as directed by the error handler

Control of flow when errors/messages occur


error handler
application code
DB–Library
DB–Library errors (dberrhandle)
SQL Server errors and messages (dbmsghandle)
DB–Library
exit
exit

- **Returns from routine installed by dbmsghandle**

DBSAVE	put the message in a message buffer
DBNOSAVE	throw away the message

Program always has the option of simply exiting

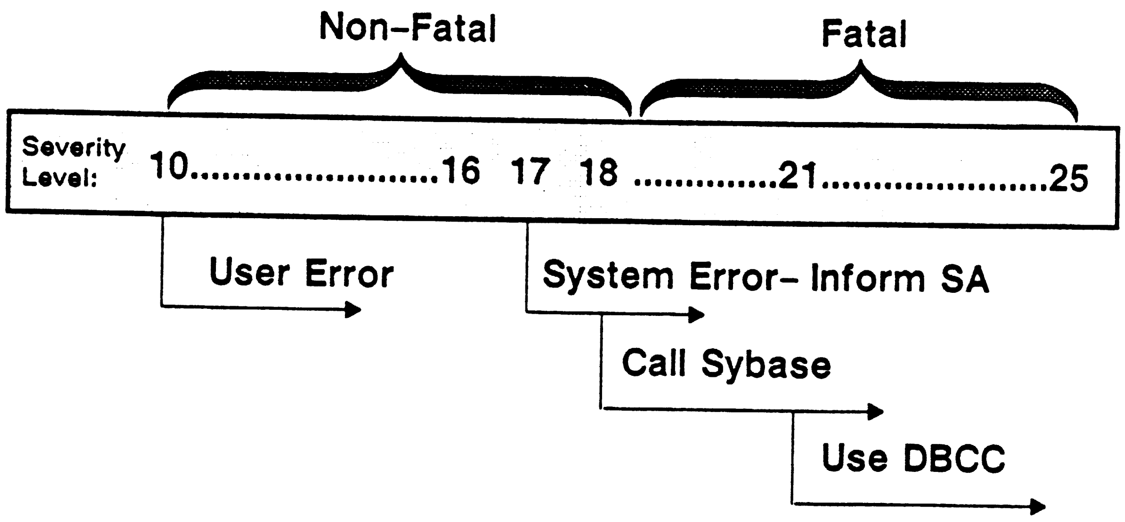
- **Returns from routine installed by dberrhandle**

INT_EXIT	print a message and abort the program
INT_CANCEL	resume processing at statement which caused the error

[illegible]

SQL Server Messages And Errors

- Categories of SQL Server error messages:



Non-Fatal
Fatal
Severity Level: 10........................16 17 18 ....21.......................25
User Error
System Error- Inform SA
Call Sybase
Use DBCC

- Recoverable messages/errors

Level 0 - Information only, such as print statement

Levels 10 -16 User Error such as syntax errors, etc.

Level 17 - Resource Error exceeding configured value, such as open databases, open objects, procedure cache, database full

Level 18 - Internal Error but user still completes work

- **Fatal Errors**

Level 19 - 23	SQL Server or Processes
Level 24	Hardware

DB–Library Errors

- **Causes**

  Incorrect use of DB–Library routines or parameters

  Internal DB–Library problems

  Operating System errors

  Errors from the SQL Server

- **Definitions for DB–Library errors**

  Documented under errors

  On–line in syberror.h, sybdb.h

- **Interaction with SQL Server Message Handler**

  If errors come from both the SQL Server and DB–Library, the SQL Server message handler is called first

  If you want to ignore the DB–Library version of the SQL Server message, check for the message number SYBESMSG

Using Handlers

Error/Message	Message Handler Called?	Error Handler Called?
SQL Syntax	Yes	Yes (SYBESMSG) *2
SQL print statement	Yes	No
SQL raiserror	Yes	No
SQL Server dies	No	Yes (SYBESEOF) *1
Timeout from SQL Server	No	Yes (SYBETIME) *3
Deadlock on Query	Yes *4	No
Timeout on Login	No	Yes (SYBEFCON) *3
Login Fails (dbopen)	Yes	Yes (SYBEPWD) *1
Use database message	Yes *2	No
Incorrect use of DB-Library calls (ie., not calling dbresults when required)	No	Yes (SYBERPND, ...)
Fatal SQL Server Error (Severity > 16)	Yes *1	Yes (SYBESMSG)

*1: Code handler to exit

*2: Code handler to ignore

*3: Code handler to continue

*4: Code handler to check for this

Installing Error Handlers

Types of Errors or Messages	Example in Class	Install With
DB-Library Errors	err_handler	dberrhandle(err_handler)
SQL Server Messages and Errors	msg_handler	dbmsghandle(msg_handler)

- **Install in the main program**

  Generally these statements are used in the beginning of the program

  You could have several error handlers and dynamically install them by using these statements

- **Linking**

  Be sure to include the error handlers in your commands to load your program

SQL Server Message Handler - C

- **Used to trap SQL Server errors and messages**
- **In the Main module**

```
extern int msg_handler( );
main( )
{
        dbmsghandle(msg_handler)
```

- **Handler code**

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
int msg_handler(dbproc,msgno,msgstate, severity, msgtext)
DBPROCESS       *dbproc;
int             msgno, msgstate, severity;
char            *msgtext;
{
        /*  msg 5701 is just a use database message, skip it.  */
        if (msgno != 5701)
                printf("Dataserver message %d, severity %d:\n\t%s\n",
                                msgno,severity,msgtext);
        if (severity > 16)
        {
                printf("Fatal SQL Server Error!  Aborting!!\n");
                dbexit( );
                exit(ERREXIT);
        }
        return (DBNOSAVE);
}
```

SQL Server Message Handler - Fortran

- **In the Main module**

```
external msg_handler
...
call fdbmsghandle(msg_handler)
```

- **Handler code**

```
INTEGER*4 FUNCTION msg_handler(dbproc,msgno,msgstate, severity,
2                                            msgtext)
INCLUDE '(fsybdb)'
INTEGER*4  dbproc, msgno, msgstate, severity
CHAR*80   msgtext

C     msg 5701 is just a use database message, skip it.
if (msgno .ne. 5701) then
      type *,'SQL Server message', msgno, 'State', msgstate,
2              'severity', severity, msgtext
end if

if (severity .gt. 16) then
      type *, 'Fatal SQL Server Error!! Aborting!!'
      call fdbexit( )
      call exit
end if

msg_handler = DBNOSAVE
return
END
```

DB-Library Error Handler - C

- **Used to trap DB-Library errors**
- **In the Main module**

```
extern int err_handler( )
main
{
      dberrhandle(err_handler);
```

- **Handler code**

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
int err_handler(dbproc, severity, errno, oserr);
DBPROCESS  *dbproc;
int severity, errno, oserr;
{
      if ((dbproc == NULL) || (DBDEAD(dbproc)))
            return (INT_EXIT);

      if (errno == SYBESMSG)
            return(INT_CANCEL);

      printf ("DB-LIBRARY error: \n\t%s\n", dberrstr(errno));   zit [illegible] [illegible] in err_handle
      if (oserr != DBNOERR)
            printf ("operating system error:\n\t%s\n",
                        dboserrstr(oserrno));
      return (INT_CANCEL);      zit [illegible] [illegible] in err_handler
}
```

DB–Library Error Handler – Fortran

- **In the Main module**

```
external err_handler
...
call fdberrhandle(err_handler)
```

- **Handler code**

```
INCLUDE '(fsybdb)'
INTEGER*4 FUNCTION err_handler(dbproc, severity, errno, oserrno)
INTEGER*4      dbproc, errno, oserrno, severity,
CHAR*(80)      message

if ((dbproc .eq. NULL) .or. (fdbdead(dbproc)) then
     err_handler =  INT_EXIT
     return
end if
if (errno .eq. SYBESMSG) then
     err_handler = INT_CANCEL
     return
end if
call fdberrstr(errno, message)
type *,'DB-LIBRARY error: ', message
if (oserr .ne.  DBNOERR) then
     call fdboserrstr(oserrno, message)
     type *,'Operating System error: ', message
end if
err_handler = INT_CANCEL
return
END
```

Points to Ponder

- **Run time SQL errors**

  If there is more than one command in the buffer, and one of the commands has a run–time error :

  dbsqlexec( ) ... returns SUCCEED

  dbresults ( ) ... returns FAIL only for the command in error

  If there is only one command in the buffer, and it has a run time error:

  dbsqlexec( ) ... returns FAIL

- **SQL Server Messages, such as a print from a procedure**

  If there is more than one command in the buffer, and one of the commands causes a SQL Server message:

  The message handler is called during dbresults( )

  If there is only one command in the buffer, and it causes a SQL Server message:

  The message handler is called during dbsqlexec( )

Try It Now!

Installing your error handlers

In the interest of time, a copy of the error handlers as shown in the foils is found on line.

- On Unix, copy **/usr/u/train/dblib/errorhandlers.c** into your directory
- On VMS, copy **SYBASE$SYSTEM:[SYBASE.TRAIN]ERRORHANDLERS.FOR.**

1. Verify and modify if necessary
   - that the SQL Server Message Handler exits the program for any error with severity greater than or equal to 16, printing a message that it is doing so on the terminal.
   - that the DB-Library Error Handler ignores SYBESMSG, which means a general SQL Server message.

2. Install the sample error handlers into your basic program
   - Use any program created in the last lab.
   - Incorporate the proper commands into the **make** files.
   - Test the error handlers by sending an invalid SQL command from your basic program.

3. Write a program which sends the following two statements to the SQL Server:

   execute sp_who

   execute sp_why (or any other non-existent stored procedure.)

   - Optional: program the code so that the error handler prints the error message, but the program code informs you which statement actually failed. (ie, was it the **dbsqlexec**?, **dbresults**?, etc.)

Make a copy of your program as it exists now, so that you have a simple template with which to begin your subsequent programs throughout the course.

Optional:

4. Modify your SQL Server message handler:
   - for any information messages (severity 0), it prints them out without any preceding information, ie: without the severity level, message number, etc.
   - test the code by creating a stored procedure in pubs which simply does a print statement. Modify one of your simple programs to send the SQL statement which runs the stored procedure.

Answers to Error Handler Lab

Problem 2

```
/* This is the same as previous labs, except that now it installs */
/* the errorhandlers err_handler and msg_handler.  See the changes */
/* in the makefile also (below) */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler( );
extern int msg_handler( );

main( )
{

        DBPROCESS     *dbproc;
        LOGINREC      *login;

        /* install the error handlers */
        dbmsghandle(msg_handler);              ← declaratic
        dberrhandle(err_handler);

        login = dblogin( );

        dbproc = dbopen(login,NULL);

        /* illegal command */
        dbcmd(dbproc," select * from publishers");
        dbcmd(dbproc," where plub_id = '1234'");

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed -- exiting program\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( dbresults(dbproc) != NO_MORE_RESULTS );
        {
                dbprrow(dbproc);
        }

        dbexit( );
        exit(STDEXIT);
```

Problem 3

```
/* Lab Number 3.2 */
/* This program installs the two sample error handlers, err_handler */
/* msg_handler, and then tests them by running an invalid SQL command */
/* Also prints out at which point the error occurred -- should be at */
/* dbsqlexec time, because the error is a syntax error */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler();
extern int msg_handler();

main( )
{

        DBPROCESS    *dbproc;
        LOGINREC     *login;
        RETCODE      results;

        /* install the error handlers */
        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin( );

        dbproc = dbopen(login,NULL);

        dbcmd(dbproc, "exec sp_who");
        /* illegal command */
        dbcmd(dbproc, " exec sp_why");

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( (results = dbresults(dbproc)) != NO_MORE_RESULTS)
        {
                if (results == FAIL)
                {
                        printf("dbresults failed\n");
                        dbexit();
                        exit(ERREXIT);
                }
                dbprrow(dbproc);
        }

        dbexit( );
        exit(STDEXIT)
```

Optional

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

int msg_handler(dbproc, msgno, msgstate, severity, msgtext)
DBPROCESS    *dbproc;
int          msgno;
int          msgstate;
int          severity;
char         *msgtext;
{
        if (msgno == 5701)
                return(DBNOSAVE);

        if (severity == 0)
        {
                printf("%s/n",msgtext);
                return(DBNOSAVE);
        }

        /* else, print all the details */

        printf("DataServer message %d, state %d,\n", msgno, msgstate);
        printf("severity %d:\n\t%s\n", severity, msgtext);

        if (severity > 16)
        {
                printf("Program Terminated!  Fatal Dataserver Error");
                dbexit( );
                exit(1);
        }

        return(DBNOSAVE);
}
```

In the main program

```
ibcmd(dbproc, "execute print_msg");
```

The stored procedure

```
reate procedure print_msg as
      print 'This is just a message'
```

SQL text with parameters

- **Syntax**

  dbfcmd (dbproc, cmdstring, arg1, arg2,...)

- **Function**

  For each occurrence of a formatting command (%) in the command string, converts and substitutes the next argument in the argument list

  Appends the command string to the command buffer

  Modeled after the C routine printf; maximum number of parameters in one call is limited to 8

  Can be freely intermixed with calls to dbcmd

- **C example**

```
char   *table = "sales";
char    *storid = "6380";
int      qty = 50;
• • •

dbfcmd(dbproc, "select * from %s", table);
dbfcmd(dbproc, "   where qty > %d", qty);
dbfcmd(dbproc, "   or stor_id = '%s' ", storid);
```

Fortran Example – fdbfcmd

- **Fortran Usage**

  Same as C except trailing blanks are deleted

- **Typical Formatting commands**

%d	INTEGER*4	(decimal number)
%s	CHARACTER*n	(character string)
%f	REAL*8	(floating point number)

- **Example**

```
CHARACTER*10    table
CHARACTER*10    storid
INTEGER*4       qty

table = 'sales'
storid = '6380'
qty = 50

call fdbfcmd(dbproc, 'select * from %s', table)
call fdbfcmd(dbproc, ' where qty > %d', qty)
call fdbfcmd(dbproc, ' or stor_id = "%s" ', storid)
```

1: exec er voor .

2: '%s' => ""%s""

3:

␣ and

t

4: bij fortran geen traili spaces .

What's wrong with these?

- **Batch 1:**

```
dbcmd ( dbproc, "sp_who");
dbcmd ( dbproc,  " sp_help authors");
```

- **Batch 2:**

```
dbcmd ( dbproc, "select au_lname from authors");
dbfcmd ( dbproc, " where name='%s'", "Bennet's");
```

- **Batch 3:**

```
dbcmd  ( dbproc, "select  *  from sales");
dbfcmd ( dbproc, "  where qty = %d", qty);
dbcmd  ( dbproc, "and stor_id = '6380'");
```

- **Batch 4 (Fortran):**

```
call fdbcmd( dbproc, 'select * from sales ')
call fdbfcmd( dbproc, 'where qty = %d', qty)
```

Selecting the Database

- **Initial State**

  The user is connected to the default database as set in **syslogins** or **sp_defaultdb**

- **Changing the database (the long way)**

```
dbcmd (dbproc, "use dbname");
dbsqlexec (dbproc);
dbresults (dbproc);
```

- **Short way:**

```
dbuse(dbproc, "dbname");
```

  Can be done at any time;

  Be aware that it uses the command buffer

Changes to Handlers for 4.0

- **New Message Handler parameters**

```
(dbproc, msgno, msgstate, severity, msgtext,
        servername, procname, line)
...
char *servername;
char *procname;
DBSMALLINT line;
```

Additional code might be:

```
    if (strlen(servername) > 0)
                printf("Server %s, ", servername);
    if (strlen(procname) > 0)
                printf("Procedure %s ", procname);
    if(line>0)
                printf("Line %d, line);
```

- **New Error Handler parameters**

```
(dbproc, severity, dberr, oserr, dberrstr, oserrstr)
...
char  *dberrstr;
char  *oserrstr;
```

Revised code would print the error strings from the parameters rather than calling dberrstr, dboserrstr.

Summary

- **Error Handlers**

  A way to invoke your own routine(s) each time an error or message occurs

  dberrhandle – Installs an handler for DB–Library errors

  dbmsghandle – Installs an handler for SQL Server messages and errors

  Both can either exit or return to the application

- **dbfcmd**

  Add text to the command buffer while specifying parameters

- **dbuse**

  Use this to change databases quickly

  Appends to your command buffer and sends it

Lab Exercises: dbfcmd

From this point on, all programs should be written to use your modified versions of the error handlers.

1. Write a program which prompts the user for an author's last name.
   - Use the input to compose a **select** statement which gets all columns for that author from the **authors** table.
   - Test the program by typing in a valid author's name (such as Bennet).
   - Test the program by typing in a name that does not exist. (Don't modify the program so that it gives an error message if the name does not exist; we'll do that in the next module.)
   - Save this program for use in the next labs.

Optional Lab (extra tricky):

2. Write a program which prompts the user for a string
   - Use that string as a password to establish a DBPROCESS as user clerkN, where N is your user number.
   - Provide code such that if the login fails, the user is told so and prompted again.
   - How will this affect the error handlers which we wrote?

Lab Answers

Problem 1

```
/* using dbfcmd */
/* this program prompts user for an author and selects */
/* from authors table based on inputted author */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler( );
extern int msg_handler( );

main( )
{

        DBPROCESS    *dbproc;
        LOGINREC     *login;
        char         author[40];

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin( );

        dbproc = dbopen(login,NULL);

        printf("Please enter an author's last name: ");
        scanf("%s",author);

        /* dbfcmd is messy, but one possible author is O'Leary */
        dbcmd(dbproc," select * from authors");
        dbfcmd(dbproc," where au_lname = \"%s\" ",author);

        if (dbsqlexec (dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while ( dbresults(dbproc) != NO_MORE_RESULTS )
        {
                dbprrow(dbproc);
        }

        dbexit( );
        exit(STDEXIT);
```

Optional

```
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
extern int err_handler( );
extern int msg_handler( );
main( )
{
        DBPROCESS    *dbproc;
        LOGINREC     *login;
        char         author[40], password[10];
        dbmsghandle(msg_handler);
        dberrhandle(err_handler);
        login = dblogin( );
        DBSETLUSER(login, "clerkN");

        /* get password first time */
        printf("Please enter a password: ");
        scanf("%s", password);
        DBSETLPWD(login, password);
        dbproc = dbopen(login, NULL);

        while (dbproc == NULL)
        {
                printf("Please enter password again: ");
                scanf("%s", password);
                DBSETLPWD(login, password);
                dbproc = dbopen(login,NULL);
        }

        printf("Please enter an author's last name: ");
        scanf("%s",author);

        dbcmd(dbproc," select * from authors");
        dbfcmd(dbproc," where au_lname = \"%s\"",author);
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }

        while (dbresults(dbproc) != NO_MORE_RESULTS)
        {
                dbprrow(dbproc);
        }

        dbexit( );
        exit(ERREXIT)
```

Change in DB-Library error handler to make this work:

```
/* this is the traditional DB-Library error handler routine, except it */
/* does not exit if it's a bad password*/

#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

int err_handler(dbproc, severity, errno, oserr);
DBPROCESS    *dbproc;
int          severity, errno, oserr;
{
             if ( (dbproc == NULL) || (DBDEAD(dbproc)) )
        {
             if (errno == SYBEPWD)        /* bad login/password */
             {
                    /* don't exit in this case */
                    printf("DB-LIBRARY  error:\n\t%s\n",dberrstr(errno));
                    return (INT_CANCEL);
             }
             else
                    return (INT_EXIT);
        }
        if (errno == SYBESMSG)
                return(INT_CANCEL);

        printf ("DB-LIBRARY error: \n\t%s\n", dberrstr(errno));
        if (oserr != DBNOERR)
                printf ("operating system error:\n\t%s\n",
                               dboserrstr(oserrno));
        return (INT_CANCEL);
```

SYBASE

Module 4
Processing Results

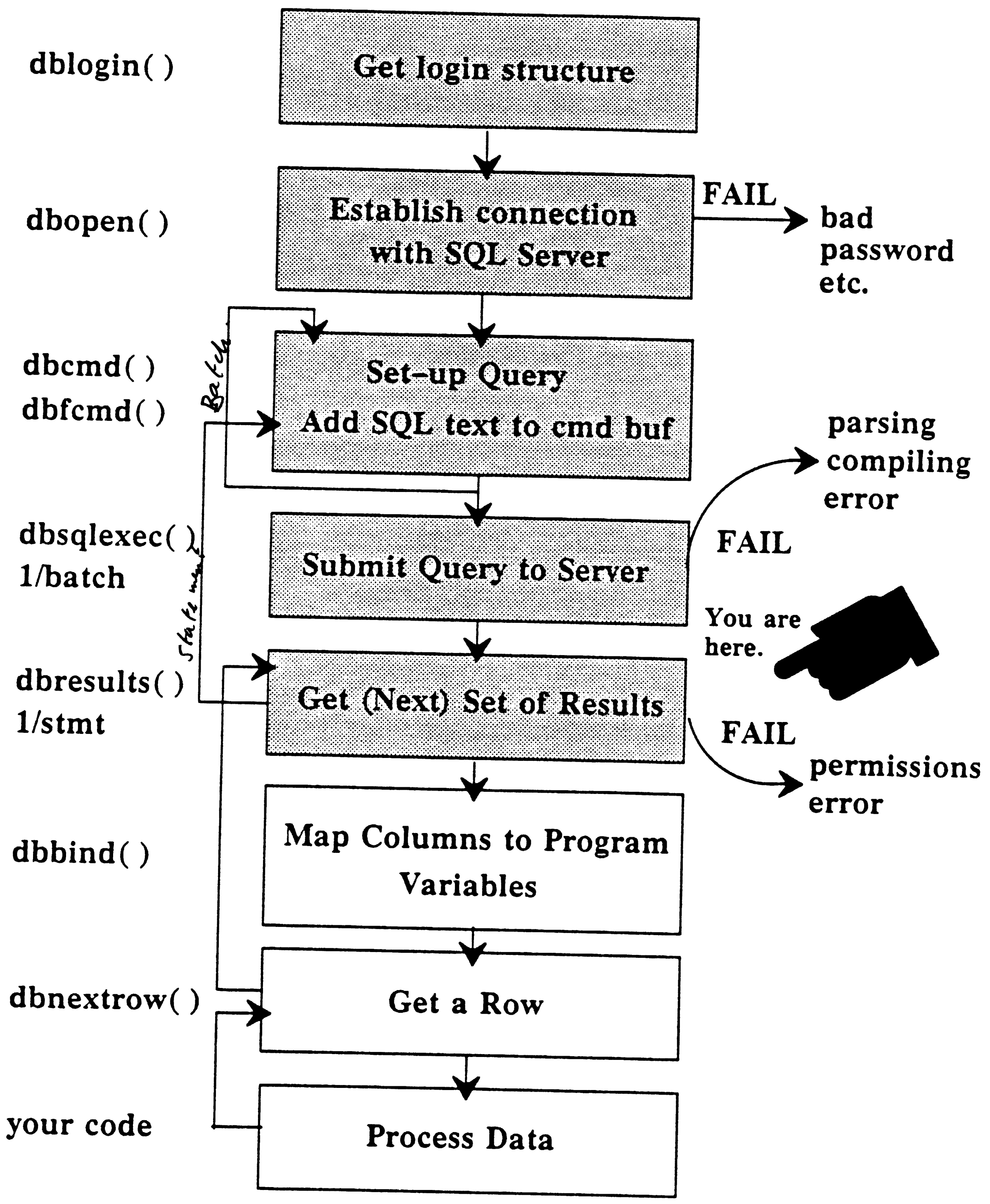
Objectives

- Establish a link between the incoming data and program variables

- Convert data from the data type in the database to any other datatype
- Understand the flow of data from the SQL Server to the application

- Test for the presence or absence of incoming data

Overview


dblogin( )
Get login structure
dbopen( )
Establish connection with SQL Server
FAIL
bad password etc.
dbcmd( )
dbfcmd( )
Batch
Set-up Query
Add SQL text to cmd buf
parsing compiling error
FAIL
dbsqlexec( )
1/batch
Statement
Submit Query to Server
You are here.
dbresults( )
1/stmt
Get (Next) Set of Results
FAIL
permissions error
dbbind( )
Map Columns to Program Variables
dbnextrow( )
Get a Row
your code
Process Data

dbresults - again

- **Function of dbresults**

  Builds a description of what is coming back from the server

  Must be called once per each SQL statement in the batch

  : maak er een loop van.

- **Implications for stored procedures**

  dbresults must be called once for each select in a stored procedure which returns data

- **Implications for triggers**

  Normally updates, etc., which don't return data do not require a call to dbresults

  If a trigger on an update or delete did a select, instead of using print, the program might fail because of unexpected results

- **Returns from dbresults**

  SUCCEED = data may be available from selects

  FAIL = permission violation on that command

  NO_MORE_RESULTS = no more SQL statements in the batch or no more selects in the stored procedure

Programming technique with dbresults

- **In most cases, use a loop**

C:

```
while ( (results = dbresults(dbproc)) !=
                            NO_MORE_RESULTS)
{
     if (results != FAIL)
               /* process the rows */
}
```

Fortran:

```
results = fdbresults(dbproc)
do while (results .ne. NO_MORE_RESULTS)
     if ( results .ne. FAIL) then
C                  process the rows
     end if
     results = fdbresults(dbproc)
end do
```

- **Notes**

Always check for FAIL, since this indicates a run-time error for any SQL statement. Exception: if there is only one command in the batch.

If you absolutely know the batch only contains one statement, then a loop with dbresults could safely be replaced by a single call to dbresults.

C Example – Stored Procedure processing

- **SQL procedure:**

```
create procedure mytwoselect as
select * from dbo.sales
select * from dbo.stores
```

- **C program to get results**

```
/* include statements omitted from this excerpt */
extern int err_handler( );
extern int msg_handler( );
main( )
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        RETCODE  results;
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin( );
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, "execute mytwoselect");
        dbcmd(dbproc, " select * from master.dbo.syslogins");
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("error in dbsqlexec\n");
                dbexit( );
                exit(ERREXIT);
        }
        while ((results = dbresults(dbproc)) != NO_MORE_RESULTS)
        {
                if (results != FAIL)
                        dbprrow(dbproc);
                else printf("dbresults failed\n");
        }
        dbexit( );
        exit(STDEXIT);
}
```

Fortran Example

- **Program to get results**

```
program Lab
include '(fsybdb)'

INTEGER*4       dbproc
INTEGER*4       login
INTEGER*4       results
EXTERNAL        err_handler, msg_handler

call fdberrhandle(err_handler)
call fdbmsghandle(msg_handler)
login = fdblogin( )
dbproc = fdbopen (login, NULL)

call fdbcmd(dbproc, 'execute mytwoselect')
call fdbcmd(dbproc, ' select * from syslogins')
if (fdbsqlexec(dbproc) .eq. FAIL) then
      type *, 'fdbsqlexec failed'
      call fdbexit( )
      call exit
end if

results = fdbresults(dbproc)
do while ( results .ne. NO_MORE_RESULTS )
          if (results .ne. FAIL) then
                call fdbprrow(dbproc)
          else
                type *, 'fdbresults failed'
          results = fdbresults(dbproc)
end do

call fdbexit( )
call exit
END
```

dbbind( ): A Strategy for Processing Data

- **Copy data into program variables**

  dbbind (...) for each column to be copied

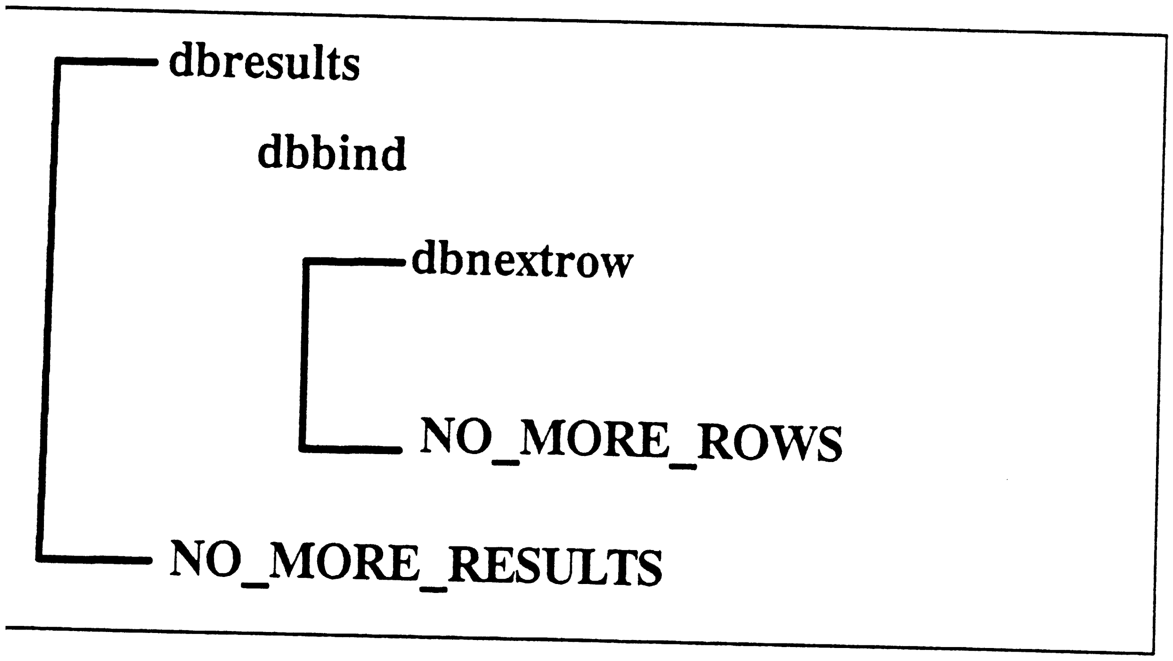
  dbnextrow(...) for each row of data returned

- **How it works**

  dbbind sets up the conversion and linkage to variables

  Each call to dbnextrow copies the next set of data into the variables

- **Placement of dbbind in your program**



dbbind( )

- **Usage**

  Always called after dbresults and before dbnextrow

  Called once for each column to be bound to a variable

  You do not have to bind all columns

  Data is not moved into the variables until you call dbnextrow

  Some data type conversions are not legal

- **Syntax**

  maximale lengte
  ↓

  dbbind(dbproc, col#, var_type,var_length, var_address)

  Only used for regular (non-compute) results

- **Column number**

  An integer corresponding to the results of the select :

select au_lname, zip ...:	au_lname is col 1
select zip, au_lname...:	au_lname is col 2

dbbind Parameters (cont)

dbbind(dbproc, col#, var_type,var_length, var_address)

- **Variable type**

  Represents the type of the <u>program variable</u>, not the type of the database column

  If the program variable type differs from the database type, and conversion is legal, conversion will be done

  Legal variable types are listed in the documentation under dbbind

- **Variable length**

  Can be 0 as long as you are sure the program variable is large enough to hold the converted data

- **Variable address**

  A common mistake in C is to forget to make it an address (&variable)

  For strings, simply give the string name

  In Fortran, there is no distinction

Data Conversion Using dbbind

dbbind(dbproc, col#, var_type,var_length, var_address)

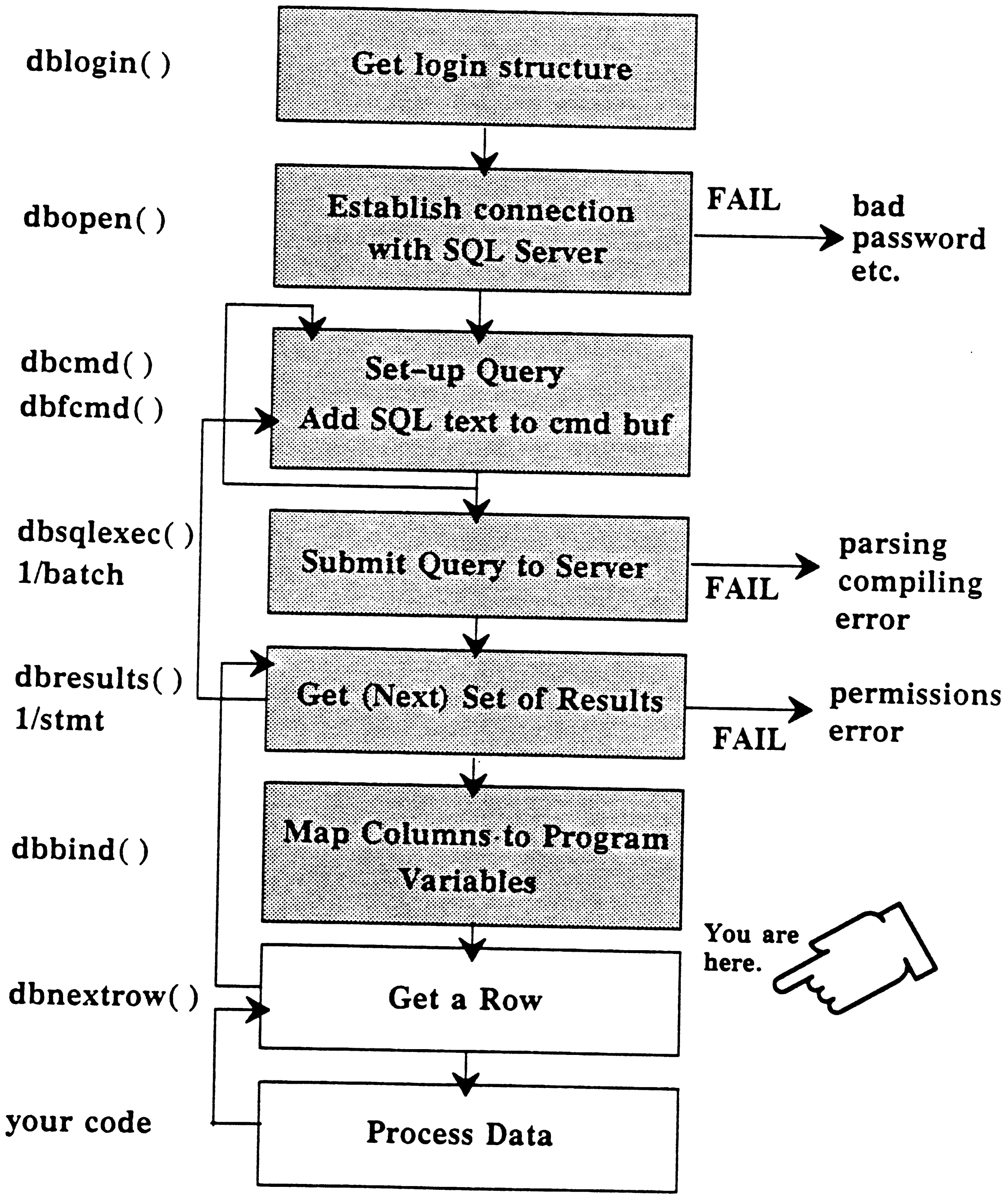
var_type in dbbind	C program datatype	Fortran program datatype	SQL Server datatype
TINYBIND	DBTINYINT	LOGICAL*1	SYBINT1
SMALLBIND	DBSMALLINT	INTEGER*2	SYBINT2
INTBIND	DBINT	INTEGER*4	SYBINT4
CHARBIND	DBCHAR	CHARACTER*(*)	SYBCHAR
STRINGBIND	DBCHAR	CHARACTER*(*)	SYBCHAR
NTBSTRINGBIND	DBCHAR	CHARACTER*(*)	SYBCHAR
VARYCHARBIND	DBVARYCHAR	RECORD /VARYCHAR/	SYBCHAR
BINARYBIND	DBBINARY	CHARACTER*(*)	SYBBINARY
BITBIND	DBBIT	LOGICAL*1	SYBBIT
DATETIMEBIND	DBDATETIME	CHARACTER*8	SYBDATETIME
MONEYBIND	DBMONEY	CHARACTER*8	SYBMONEY
FLT8BIND	DBFLT8	REAL*8	SYBFLT8
VARYBINBIND	DBVARYBIN	RECORD /VARYBIN/	SYBBINARY

- Notes

var_type must match the program variable type

If var_type and the program variable don't match the SQL Server type, conversion is done

Where are we now?



Processing the Rows

- **dbnextrow (dbproc)**

  Must be called once for each row, unless you cancel the remaining rows

  Makes the next set of rows available to the program

  If dbbind was used, copies the data to the variables (for regular, non–compute, rows)

  The previous row of data is no longer available in this mode of operation

- **Returns**

NO_MORE_ROWS	
REG_ROW	(normal results of a select)
FAIL	major error such as network or dataserver failed

  (Two other returns are discussed in next module)

  Declared as type STATUS, not RETCODE

- **Usage**

```
C: while (dbnextrow(dbproc) != NO_MORE_ROWS)
       do something with the data
```

Fortran:

```
do while (fdbnextrow(dbproc) .ne. NO_MORE_ROWS)
     do something with the data
end do
```

A C Example of **dbbind** and **dbnextrow**

- Example

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
extern int err_handler( );
extern int msg_handler( );
main( )
{
        DBPROCESS      *dbproc;
        LOGINREC       *login;
        RETCODE        results;
        DBCHAR         storeid[5];
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin( );
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, "select stor_id from stores");
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed");
                dbexit( );
                exit(ERREXIT);
        }
        dbresults(dbproc);
        dbbind (dbproc, 1, STRINGBIND, 0, storeid);
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
        {
                printf ("String bind  %s\n", storeid);
        }
        dbexit( );
        exit(STDEXIT);
}
```

A Fortran Example of **dbbind** and **dbnextrow**

- **Example**

```
program BindProg
include '(fsybdb)'
INTEGER*4       dbproc
INTEGER*4       login
INTEGER*4       results
CHARACTER*4     storeid
EXTERNAL        err_handler, msg_handler

call fdberrhandle(err_handler)
call fdbmsghandle(msg_handler)
login = fdblogin( )
dbproc = fdbopen (login, NULL)

call fdbcmd(dbproc, 'select stor_id from stores')
if (fdbsqlexec(dbproc) .eq. FAIL) then
      type *, 'fdbsqlexec failed'
      call fdbexit( )
      call exit
end if

call fdbresults(dbproc)
call fdbbind(dbproc, 1, CHARBIND, 0, storeid)
do while ( fdbnextrow(dbproc) .ne. NO_MORE_ROWS)
      type *, 'Char bind', storeid
end do

call fdbexit( )
call exit
END
```

More dbbind Examples

- **Binding stor_id which is CHAR(4):**

```
DBCHAR   store[5]
dbbind (dbproc,1,STRINGBIND,0,store)
```

or...

```
 /* this does a conversion*/
DBINT   store
dbbind (dbproc,1,INTBIND,0,&store)
```

- **Common ways of binding character data**

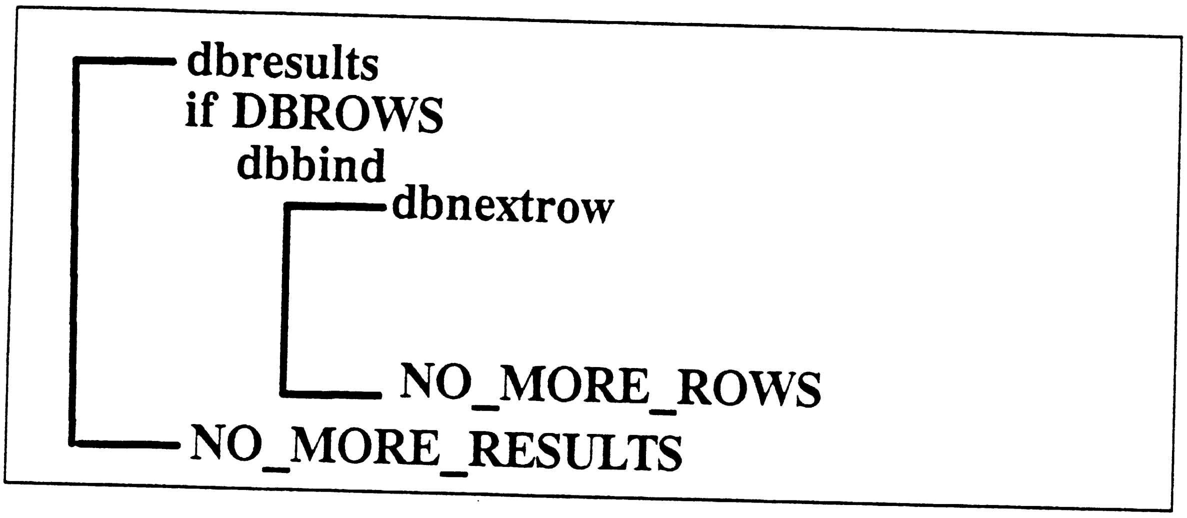
STRINGBIND	blank padding	null terminator
NTBSTRINGBIND	no blanks	null terminator
CHARBIND	blank padding	no null terminator

- **What's a better way than this?**

```
DBCHAR      au_lname[41];
. . .
while (dbresults(dbproc)==SUCCEED)
     dbbind(dbproc,1,CHARBIND,0,au_lname);
     au_lname[40] = '\0';
while (dbnextrow(dbproc) != NO_MORE_ROWS)
     printf(" Last name:  %s\n",au_lname);
```

DBROWS: What really came back?

- Placement of DBROWS in your program



- To check for rows from a select:

C:

```
dbresults (...)
if (DBROWS(dbproc) == SUCCEED)
     there are rows
```

Fortran:

```
call fdbresults (...)
return_code = fdbrows(dbproc)
if (return_code .eq. SUCCEED) then
     there are rows
```

Other useful functions

- **Getting rid of data**

  dbcancel (dbproc)

  Cancel and flush the results of the current command $\boxed{\text{batch}}$

  You may safely do a new dbsqlexec after this

  dbcanquery(dbproc)

  Cancel the results of the current $\boxed{\text{select}}$

  You may safely do a new dbresults after this

  This is faster than doing dbnextrow to the end

- **Dealing with Nulls**

  dbsetnull(dbproc, bindtype, bindlen, bindval)

  Tells dbbind what to use for NULL values for the various data types.

  If dbsetnull is not used, there are default values for NULLS. See the documentation for dbsetnull.

Summary

- **dbbind**

  Binds regular column data to program variables

  Optionally does conversion of data types

- **dbnextrow**

  Gets the next (or first) row of data

  Copies the data to the bound variables

- **DBROWS**

  Tells you if any rows were returned

- **dbcancel, dbcanquery**

  Cancels batch or results of current select

Putting it together

- Framework for the lab exercise – updating database
  - ∆ Open the connection
  - ∆ Build and send a select (**dbfcmd**, **dbsqlexec**)

    Prompt the user for a title id

    Use it to select title and price from pubs.dbo.titles
  - ∆ Get results (**dbresults**, **dbbind**, **dbnextrow**)

    Bind the title and price into variables

    Get each row of data (should be only one)
  - ∆ Process the data

    Print out the title and the current price

    Let the user input a new price

    Update the database with the new price (dbfcmd, dbsqlexec)
  - ∆ Close up and exit (**dbexit**)

Summary

- **K.I.S.S. – keep it simple . . .**

  dblogin
  dbopen, dbexit
  dbcmd, dbfcmd
  dbsqlexec, dbresults, dbnextrow
  dbbind
  dbmsghandle, dberrhandle

- **Have Fun!**

Advanced Topics

- Row Buffering (dbgetrow)
- Browse Mode (dbqual, dbtsput)
- Text Data (dbwritetext)
- Using RPC protocol (dbrpc...)
- Return values/ return status from procedures (dbret...)
- Handling compute data (dbalt...)
- Programming for Ad Hoc queries
- Access results data using dbdata
- Use conversion routines to change data types (dbconvert)
- Debugging Techniques
- Bulk Copy(bcp_...)
- Two–Phase Commit (..._xact)

Lab Exercise: Processing Results using dbbind

1. Write a program which prompts the user for a title-id.

   - Retrieve the title and the price from the pubs database table titles.
   - Display the title and price on the screen without using dbprrow.
   - Ask the user for a new price.
   - Update the database with the new price and exit.
   - Run the program and test it by using ISQL after you run it to verify that the price got changed.

2. Modify the program to check to see if the title-id selected returned any rows. If not, print an error message and exit.

Optional, but highly recommended:

3. Modify the program above

   - Display a list of title-ids and the associated titles before asking the user for input.
   - Experiment with displaying the titles using STRINGBIND versus NTBSTRINGBIND.
   - Don't worry about making the lines wrap nicely on the screen, or if you don't like ugly output, set up the program so that only the first 25 characters are returned.

Optional:

4. Make a stored procedure which takes as a parameter the title-id.

   - Let it determine if the title-id exists and if not, use a print to send a message. Otherwise, it should do the select for you.
   - Modify the previous programs to call the stored procedure with the parameter instead of building the select.

Pseudocode for labs

Lab 4.1

```
include statements, declare and install error handlers
set up DBPROCESS

get title_id from user

build and execute the select; if failed, error and exit
do a dbresults

do the binds

set up dbnextrow loop and print out each row

get new price from user

build and execute the update; if failed, inform user

close and exit
```

Lab 4.2

```
include statements, declare and install error handlers
set up DBPROCESS

get title_id from user

build and execute the select; if failed, error and exit
do a dbresults

if no rows returned, error and exit

-- insert rest of 4.1 pseudocode here --
```

Lab 4.3

```
include statements, declare and install error handlers
set up DBPROCESS

build and execute a select to get all title_ids and associated titles; if failed, error and exit
do a dbresults

bind the title_id and the title

set up dbnextrow loop and print out each row

get title_id from user

-- insert rest of 4.2 pseudocode here --
```

Lab Answers

Problem 1

```
/* Lab Number 4.1 */
/* prompt for title_id, retrieve and print title and price of */
/* corresponding book, ask for new price, update the price */

/*  include statements, declare and install error handlers set up DBPROCESS */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler();
extern int msg_handler();

main()
{

        DBPROCESS    *dbproc;
        LOGINREC     *login;
        RETCODE      return_code;

        DBCHAR       title_id[7];     /* one longer than tid length */
        DBCHAR       title[81];       /* one longer than title max length */
        DBFLT8       price;
        char         newprice[10];

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin();

        dbproc = dbopen(login,NULL);

        /* get title_id from user */
        printf("Please enter a title-id: ");
        scanf("%s",title_id);

        /*  build and execute the select; if failed, error and exit */
        dbcmd(dbproc," select title, price from titles");
        dbfcmd(dbproc," where title_id = '%s'", title_id);

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("error in dbsqlexec\n");
                dbexit( );
                exit(ERREXIT);
        }
        /*do a dbresults */
        dbresults(dbproc);

        /* do the binds */
        dbbind(dbproc,1,STRINGBIND,0,title);
        dbbind(dbproc,2,FLT8BIND,0,&price);

        /* set up dbnextrow loop and print out each row */
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
        {
                printf("Title: %s\n",title);
                printf("Cost: $%.2f\n", price);
        }

        /*  get new price from user  */
        printf("Please enter new price: ");
        scanf("%s",newprice);

        dbfcmd(dbproc," update titles set price = $%s",newprice);
        dbfcmd(dbproc, " where title_id = '%s'", title_id);

        /* build and execute the update; if failed, inform user */
        if(dbsqlexec(dbproc) == FAIL)
                printf("update  unsuccessful!\n");

        /* close and exit */
        dbclose(dbproc);
        dbexit();
        exit(STDEXIT);
}
```

Problem 2

```
/* Lab Number 4.2 */
/* prompt for title_id; if valid, retrieve and print title and price of */
/* corresponding title, ask for new price, update the price */

/*  include statements, declare and install error handlers set up DBPROCESS */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler();
extern int msg_handler();

main()
{

        DBPROCESS    *dbproc;
        LOGINREC     *login;
        RETCODE      return_code;

        DBCHAR       title_id[7];    /* one longer than tid length */
        DBCHAR       title[81];      /* one longer than title max length */
        DBFLT8       price;
        char         newprice[10];

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin();

        dbproc = dbopen(login,NULL);

        /* get title_id from user */
        printf("Please enter a title-id: ");
        scanf("%s",title_id);

        /*  build and execute the select; if failed, error and exit */
        dbcmd(dbproc," select title, price from titles");
        dbfcmd(dbproc," where title_id = '%s'", title_id);

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("error in dbsqlexec\n");
                dbexit( );
                exit(ERREXIT);
        }

        /*do a dbresults */
        dbresults(dbproc);

        /* if no rows returned, error and exit */
        if (DBROWS(dbproc) != SUCCEED)
```

```
    {
            printf("Invalid title id\n");
            dbexit();
            exit(ERREXIT);
    }


    /* ... insert rest of 4.1 here ...*/

    /* do the binds */
    dbbind(dbproc,1,STRINGBIND,0,title);
    dbbind(dbproc,2,FLT8BIND,0,&price);

    /* set up dbnextrow loop and print out each row */
    while (dbnextrow(dbproc) != NO_MORE_ROWS)
    {
            printf("Title: %s\n",title);
            printf("Cost: $%.2f\n", price);
    }

    /*  get new price from user  */
    printf("Please enter new price: ");
    scanf("%s",newprice);

    dbfcmd(dbproc," update titles set price = $%s",newprice);
    dbfcmd(dbproc, " where title_id = '%s'", title_id);

    /* build and execute the update; if failed, inform user */
    if(dbsqlexec(dbproc) == FAIL)
            printf("update  unsuccessful\n");

    /* close and exit */
    dbclose(dbproc);
    dbexit();
    exit(STDEXIT);
}
```

Problem 3

```
/* Lab Number 4.3 */

/*  include statements, declare and install error handlers set up DBPROCESS */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler();
extern int msg_handler();

main()
{

        DBPROCESS    *dbproc;
        LOGINREC     *login;
        RETCODE      return_code;

        DBCHAR       title_id[7];     /* one longer than tid length */
        DBCHAR       title[81];       /* one longer than title max length */
        DBFLT8       price;
        char         newprice[10];

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin();

        dbproc = dbopen(login,NULL);

        /* build and execute a select to get all title_ids and associated titles */
        /* if failed, error and exit */
        dbcmd(dbproc," select title_id, title from titles");

        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("error in dbsqlexec\n");
                dbexit( );
                exit(ERREXIT);
        }

        /*  do a dbresults */
        dbresults(dbproc);

        /* bind the title_id and the title */
        dbbind(dbproc,1,NTBSTRINGBIND,0,title_id);
        dbbind(dbproc,2,STRINGBIND,0,title);

        /* set up dbnextrow loop and print out each row */
        printf("Here is a list of title-ids and titles\n");
```

```
while(dbnextrow(dbproc) != NO_MORE_ROWS)
{
        printf("Title-id: %s",title_id);
        printf(" Title:  %s\n\n",title);
}


/* ... insert rest of 4.2 here ... */

/* get title_id from user */
printf("Please enter a title-id: ");
scanf("%s",title_id);

/*  build and execute the select; if failed, error and exit */
dbcmd(dbproc," select title, price from titles");
dbfcmd(dbproc," where title_id = '%s'", title_id);

if (dbsqlexec(dbproc) == FAIL)
{
        printf("error in dbsqlexec\n")
        dbexit( );
        exit(ERREXIT);
}

/*do a dbresults */
dbresults(dbproc);

/* if no rows returned, error and exit */
if (DBROWS(dbproc) != SUCCEED)
{
        printf("Invalid title id\n");
        dbexit( );
        exit(ERREXIT);
}

/* do the binds */
dbbind(dbproc,1,STRINGBIND,0,title);
dbbind(dbproc,2,FLT8BIND,0,&price);

/* set up dbnextrow loop and print out each row */
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
        printf("Title: %s\n",title);
        printf("Cost: $%.2f\n", price);
}

/*  get new price from user  */
printf("Please enter new price: ");
scanf("%s",newprice);

dbfcmd(dbproc," update titles set price = $%s",newprice);
dbfcmd(dbproc, " where title_id = '%s'", title_id);
```

```
        /* build and execute the update; if failed, inform user */
        if(dbsqlexec(dbproc) == FAIL)
                printf("update  unsuccessful!\n");

        /* close and exit */
        dbclose(dbproc);
        dbexit();
        exit(STDEXIT);
}
```

Problem 4

This is just the code for the stored procedure. The changes to the main program are minor.

```
create procedure get_title_price @title_id tid as

if not exists (select * from titles
            where title_id = @title_id)
        begin
                print 'Title does not exist'
                return
        end

select title, price from titles
where title_id = @title_id
```

SYBASE

Advanced DB-Library Student Guide

Advanced ™DB-Library Course Topics

1. Overview
2. Row Buffering & Browse Mode
3. Text Data/ RPCs
4. Miscellaneous
5. Bulk Copy
6. Two-Phase Commit

SYBASE

Module 1

DB-Library Review

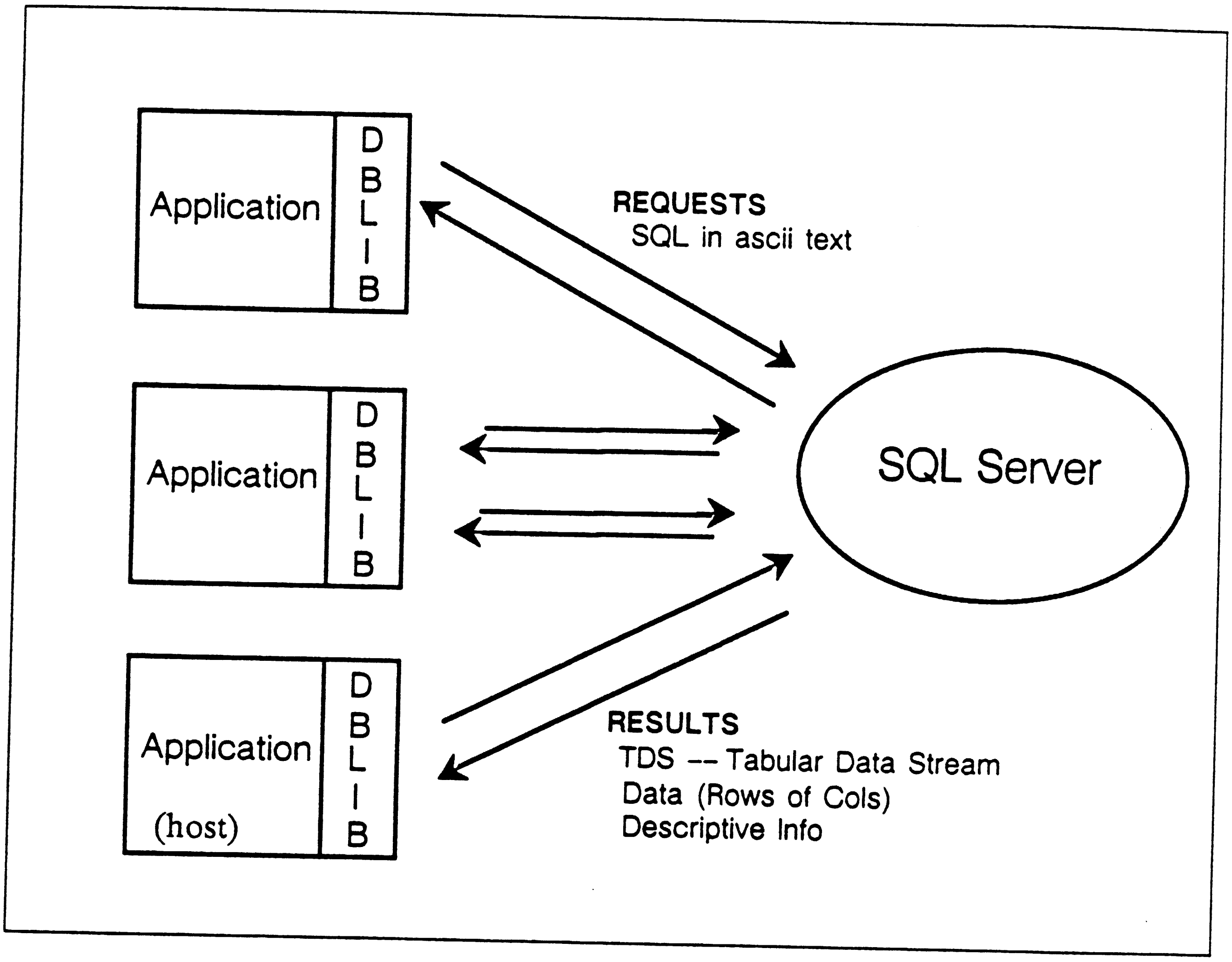
Objectives

- Understand the function of DB–Library and its relation to the SQL Server and application programs

- Review the basic calls required to execute SQL code

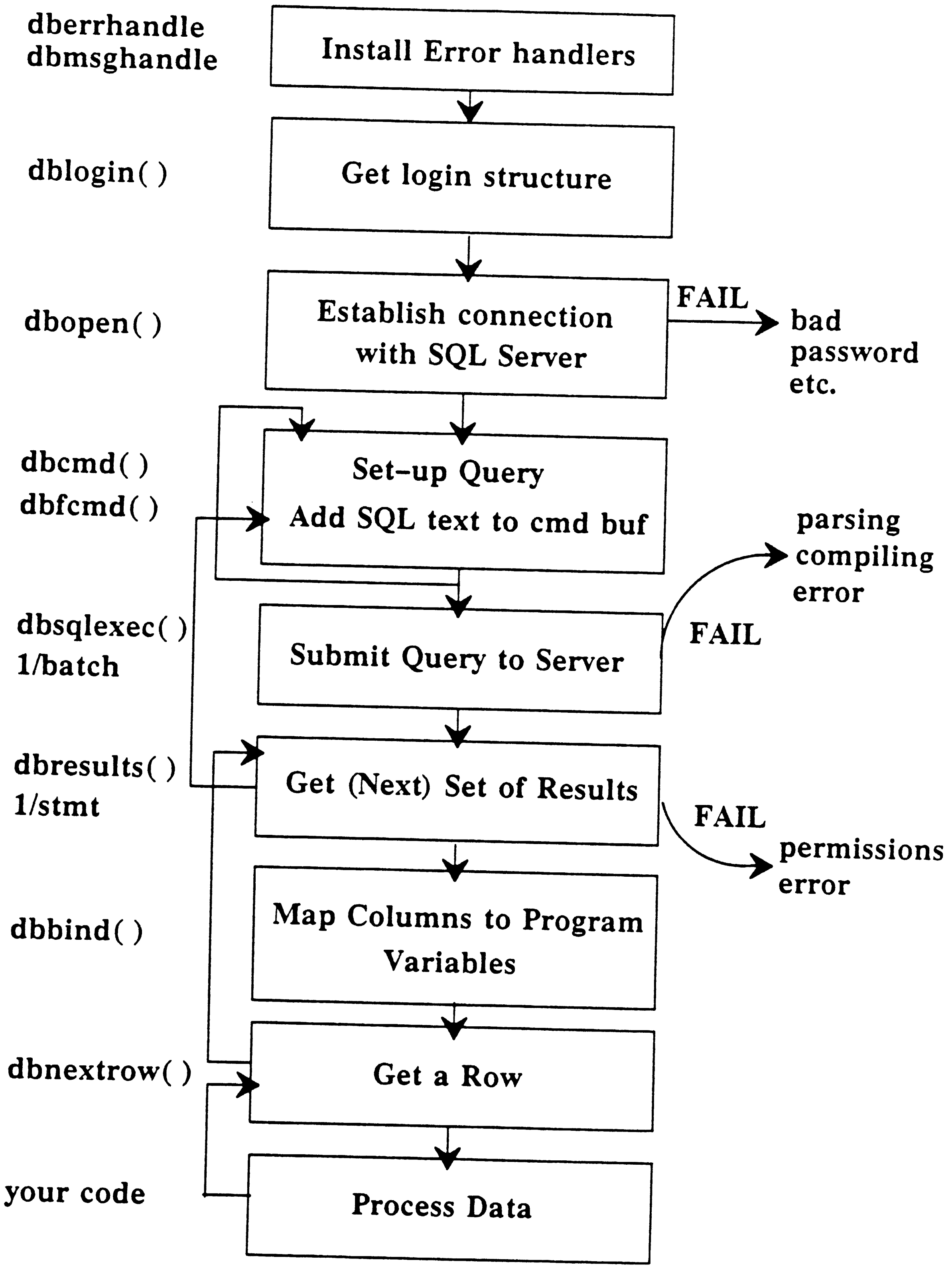
- Learn the steps required to build a "runnable" application

DB–Library and the SQL Server



- Requests to the SQL Server are in SQL text
- Results to the application are in **host** datatypes
- SQL Server uses an application protocol called 'TDS' (Tabular Data Stream)

Overview


dberrhandle
dbmsghandle
Install Error handlers
dblogin( )
Get login structure
dbopen( )
Establish connection with SQL Server
FAIL
bad password etc.
dbcmd( )
dbfcmd( )
Set-up Query
Add SQL text to cmd buf
parsing compiling error
FAIL
dbsqlexec( )
1/batch
Submit Query to Server
dbresults( )
1/stmt
Get (Next) Set of Results
FAIL
permissions error
dbbind( )
Map Columns to Program Variables
dbnextrow( )
Get a Row
your code
Process Data

Installing Error Handlers

Types of Errors or Messages	Example in Class	Install With
DB-Library Errors	err_handler	dberrhandle(err_handler)
SQL Server Messages and Errors	msg_handler	dbmsghandle(msg_handler)

- **Install in the main program**

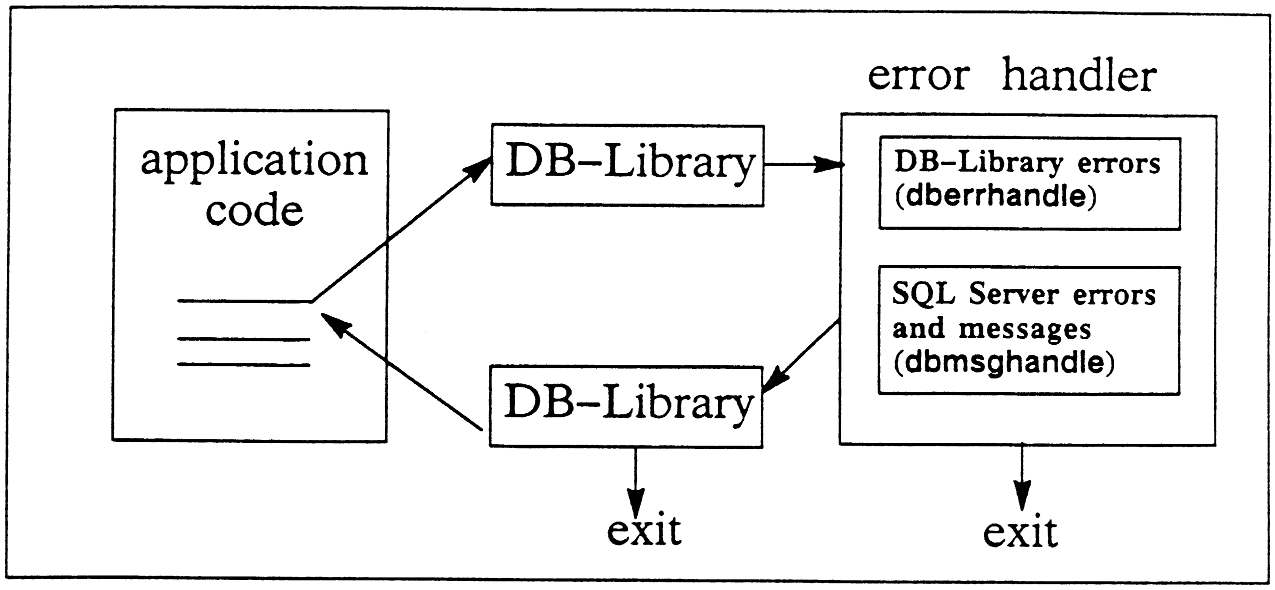
  Generally these statements are used in the beginning of the program

  You could have several error handlers and dynamically install them by using these statements

- **Linking**

  Be sure to include the error handlers in your commands to load your program

Control of flow when errors/messages occur


error handler
application code
DB-Library
DB-Library errors (dberrhandle)
SQL Server errors and messages (dbmsghandle)
DB-Library
exit
exit

- **Returns from routine installed by dberrhandle**

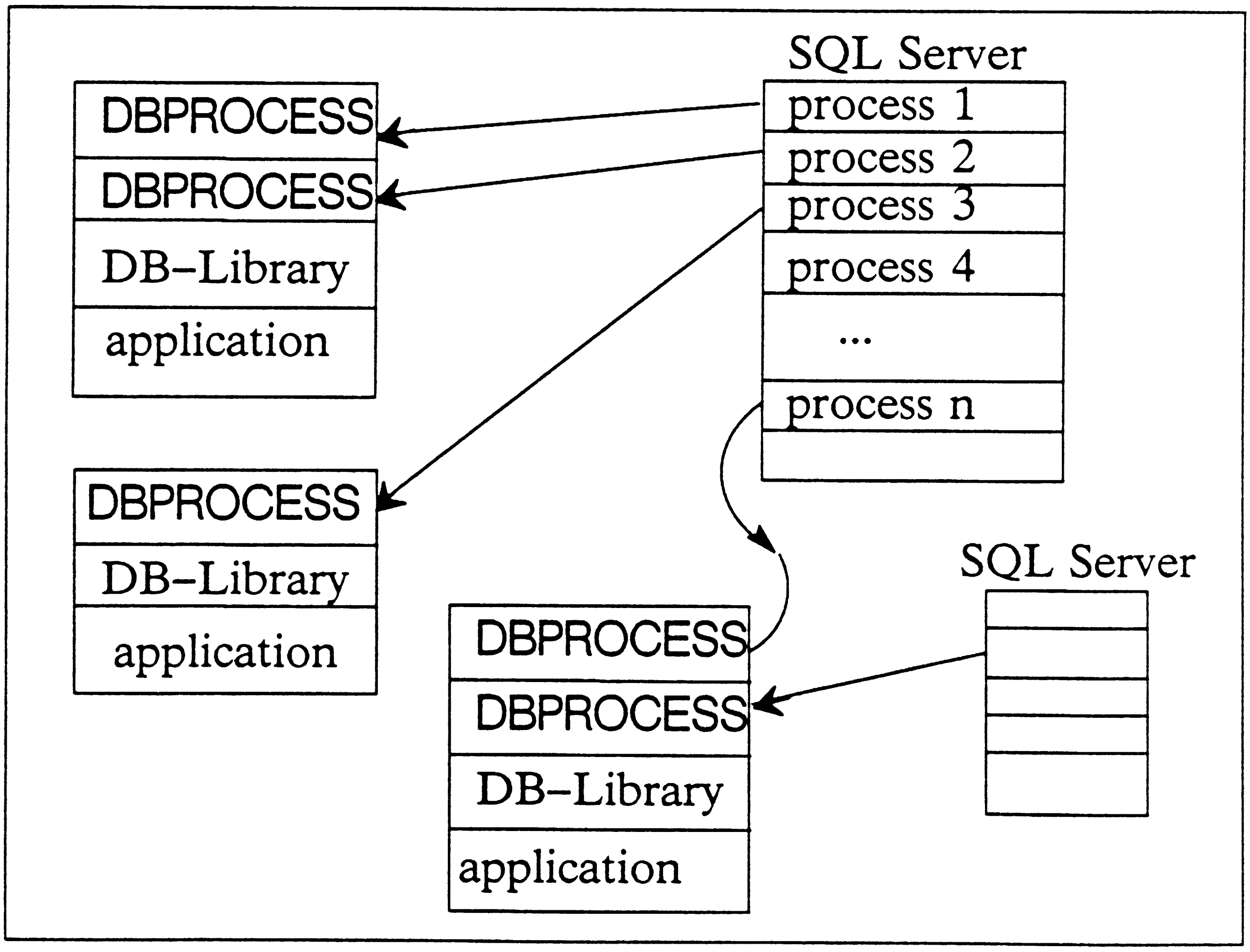
INT_EXIT	print a message and abort the program
INT_CANCEL	resume processing at statement which caused the error

- **Returns from routine installed by dbmsghandle**

Program always has the option of simply exiting, but normally will simply do a return

DBPROCESSes and the SQL Server

- For each separate DBPROCESS in an application, there is a unique server process on the SQL Server
- An application can open multiple DBPROCESSes to multiple servers, and/or multiple DBPROCESSes to one server
- Use **dblogin** to get a login structure
  Use **dbopen** to open a connection to the server



Sending SQL Commands

- **Syntax**

```
dbcmd(dbproc, cmdstring)
dbfcmd (dbproc, cmdstring, arg1, arg2,...)
dbsqlexec(dbproc)
```

- **Function of dbcmd/ dbfcmd**

Appends the command string to the command buffer

dbfcmd converts and substitutes the next argument in the argument list for each occurrence of a formatting command (%) in the command string,

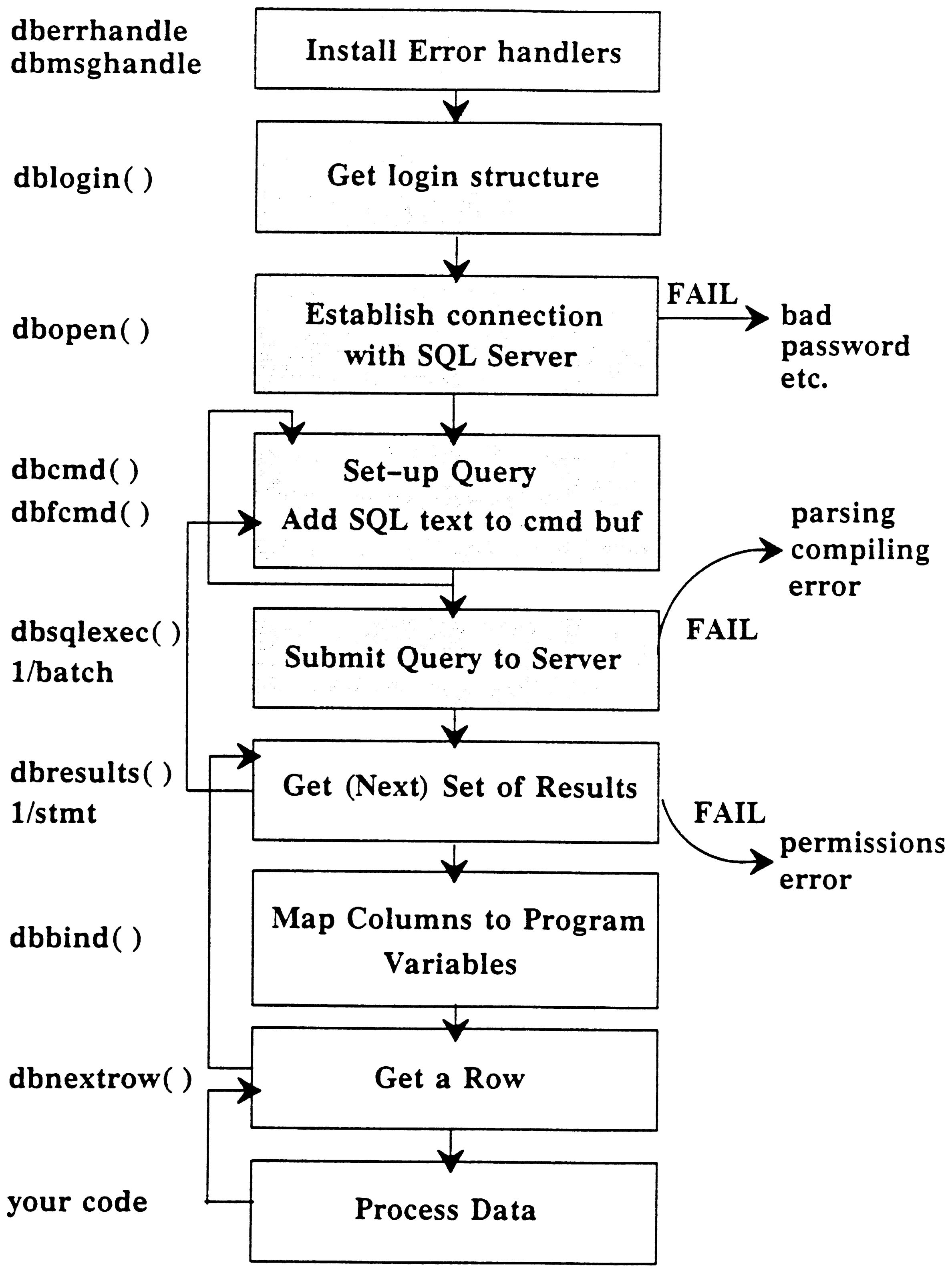
dbfcmd can be freely intermixed with calls to dbcmd

- **C example**

```
char   *table = "sales";
char    *storid = "6380";
int      qty = 50;
• • •

dbfcmd(dbproc, "select * from %s", table);
dbfcmd(dbproc, "  where qty > %d", qty);
dbfcmd(dbproc, "  or stor_id = '%s' ", storid);
dbcmd(dbproc, " select * from titles");
dbsqlexec(dbproc)
```

Where are we now?



Programming technique with dbresults

- **In most cases, use a loop**

C:

```
while ( (results = dbresults(dbproc)) !=
                        NO_MORE_RESULTS)
{
    if (results != FAIL)
            /* process the rows */
}
```

Fortran:

```
results = fdbresults(dbproc)
do while (results .ne. NO_MORE_RESULTS)
    if ( results .ne. FAIL) then
C               process the rows
    end if
    results = fdbresults(dbproc)
end do
```

- **Notes**

Always check for FAIL, since this indicates a run–time error for any SQL statement. Exception: if there is only one command in the batch.

If you absolutely know the batch only contains one statement, then a loop with dbresults could safely be replaced by a single call to dbresults.

dbbind( ): A Strategy for Processing Data

- **Establish link between data and program variables**

  dbbind (...)   for each column of returning data

  dbnextrow(...) for each row of data returned

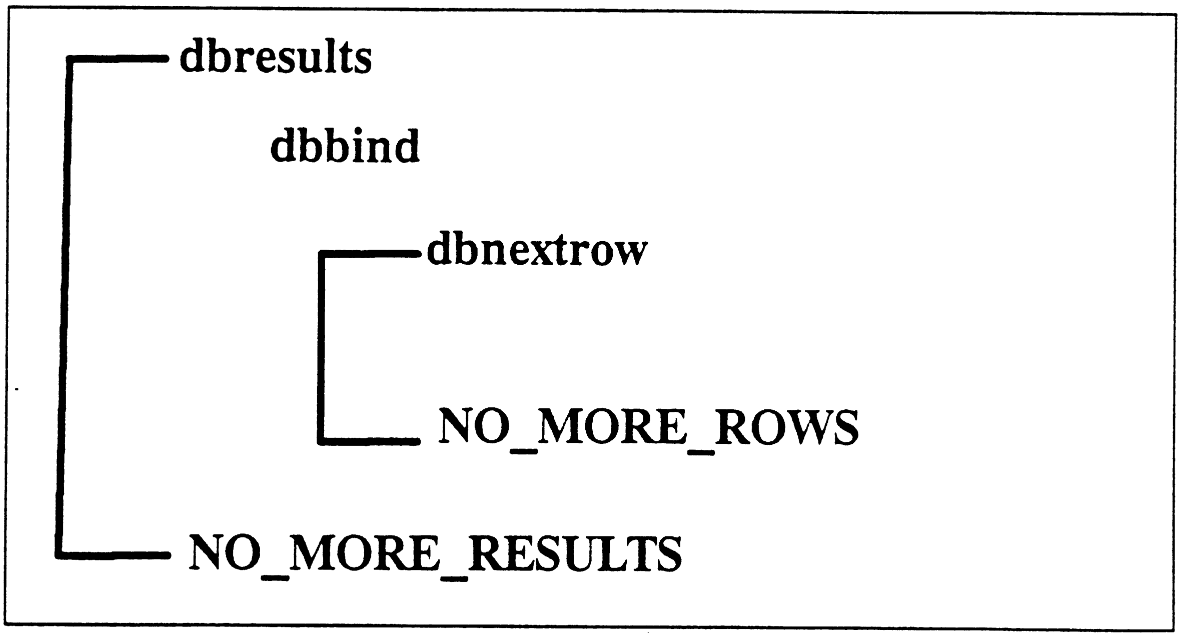
- **How it works**

  dbbind sets up the conversion and linkage to variables

  Each call to dbnextrow copies the next set of data into the variables

  It is easier to bind things like money and date to strings rather than set up date/money variables and then do conversions

- **Placement of dbbind in your program**



A C Example of **dbbind** and **dbnextrow**

- **Example**

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
extern int err_handler( );
extern int msg_handler( );
main( )
{
        DBPROCESS       *dbproc;
        LOGINREC        *login;
        RETCODE         results;
        DBCHAR          storeid[5];
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin( );
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, "select stor_id from stores");
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed");
                dbexit( );
                exit(ERREXIT);
        }
        dbresults(dbproc);
        dbbind (dbproc, 1, STRINGBIND, 0, storeid);
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
        {
                printf ("String bind  %s\n", storeid);
        }
        dbexit( );
        exit(STDEXIT);
}
```

A Fortran Example of **dbbind** and **dbnextrow**

- **Example**

```
program BindProg
include '(fsybdb)'
INTEGER*4       dbproc
INTEGER*4       login
INTEGER*4       results
CHARACTER*4     storeid
EXTERNAL        err_handler, msg_handler

call fdberrhandle(err_handler)
call fdbmsghandle(msg_handler)
login = fdblogin( )
dbproc = dbopen (login, NULL)

call fdbcmd(dbproc, 'select stor_id from stores')
if (fdbsqlexec(dbproc) .eq. FAIL) then
        type *, 'fdbsqlexec failed'
        call fdbexit( )
        call exit
end if

call fdbresults(dbproc)
call fdbbind(dbproc, 1, CHARBIND, 0, storeid)
do while ( fdbnextrow(dbproc) .ne. NO_MORE_ROWS)
        type *, 'Char bind', storeid
end do

call fdbexit( )
call exit
END
```

Putting it together

- **Framework for simple programs**
  - △ Install error and message handlers
  - △ Open the connection (**dblogin, dbopen**)
  - △ Build and send a select (**dbfcmd**, **dbsqlexec**)
  - △ Get results (**dbresults**, **dbbind**, **dbnextrow**)
  - △ Process the data
    Print out the rows
  - △ Close up and exit (**dbexit**)

DB-Library Files (Unix - C)

- **$SYBASE/include**

  Definitions are contained in header files:

sybfront.h	must be included first contains type definitions
sybdb.h	defines structures;
sybdbtokens.h	
sybloginrec.h	included automatically by sybdb
syberror.h	contains error severity definitions

- **$SYBASE/lib**

libsybdb.a	Contains the code for all the functions and macros

- **Usage**

  In your C program, begin the program with:

```
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
```

  Specify the library file when linking the program

DB-Library Files (VMS - Fortran)

- **SYBASE$SYSTEM:[SYBASE.INCLUDE]**

  Contains a header file with definitions of parameter and function return values, to be included with the Fortran program.

  All the appropriate C files are converted and combined into one text library file

  File name: FSYBINC.TLB

- **SYBASE$SYSTEM:[SYBASE.LIB]**

  Fortran programs require two Sybase link libraries, in addition to the standard system libraries:

LIBFSYBDB.OLB	provides the interface to the C library
LIBSYBDB.OLB	identical to the C library on Unix

  Libraries can be linked shareable or non-shareable

- **Usage**

  In your Fortran program, begin the program with:

```
include '(fsybdb)'
```

  Specify the libraries when linking the program

Compiling & Loading

(Unix - C)

- **Define SYBASE if necessary**

```
setenv SYBASE /usr/u/sybase/...
```

- **For compilation, add the include files from $SYBASE/include**

```
cc myprogram.c -I$SYBASE/include
```

- **For loading, add the library files from $SYBASE/lib**

```
cc myprogram.c
    -I$SYBASE/include
      $SYBASE/lib/libsybdb.a -o output
```

- **For efficiency, use a make file**

  A sample makefile is in the appendix

Compiling & Loading

(Fortran)

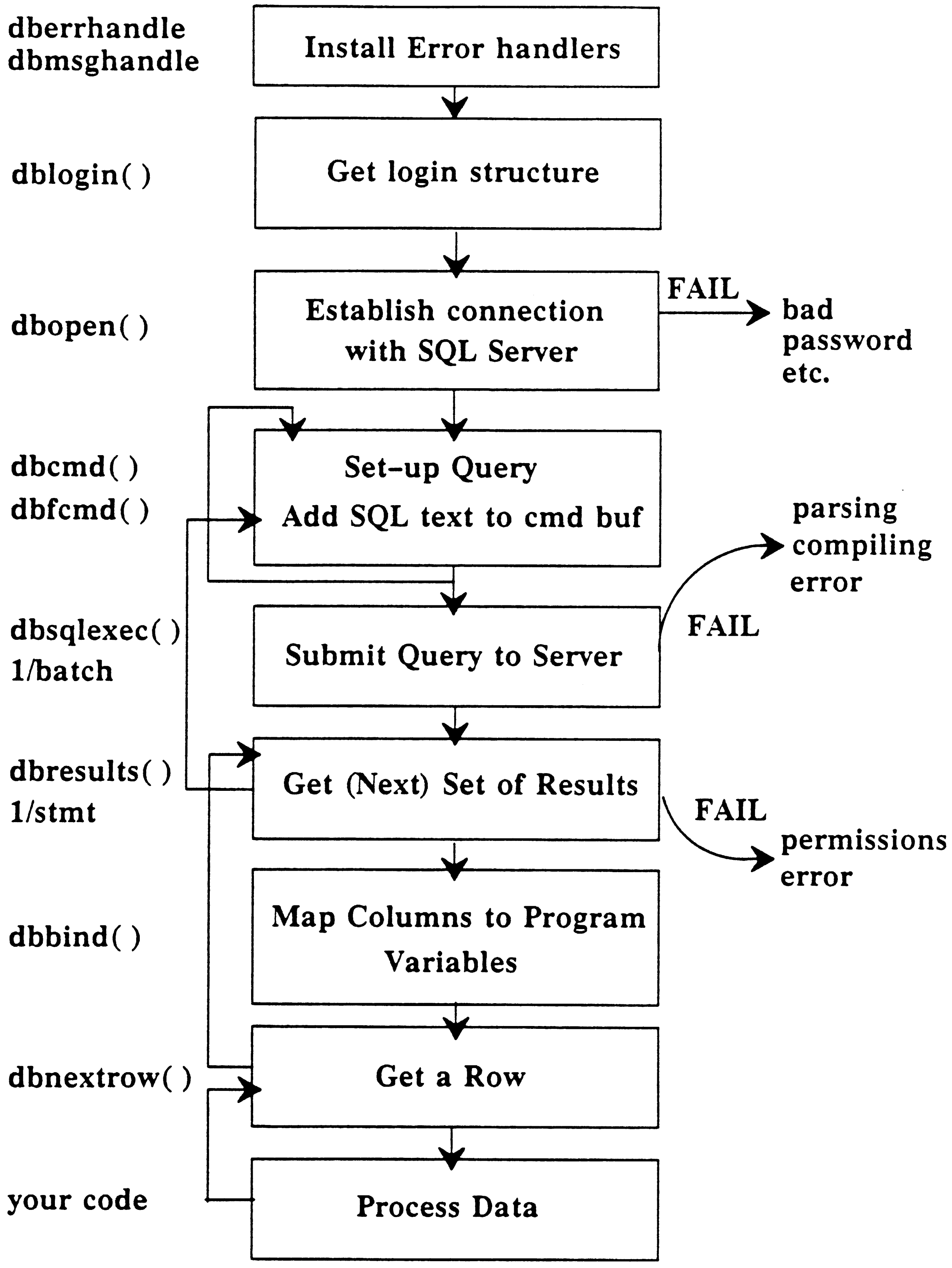
- **For compilation: (add to LOGIN.COM)**

```
$DEFINE FORT$LIBRARY
   SYBASE$SYSTEM:[SYBASE.INCLUDE]FSYBINC.TLB
FOR myprog.for /warn = dec
```

- **For linking, sharable make a LINK.COM file**

```
$ LINK myprog, -
   SYBASE$SYTEM:[SYBASE.LIB]LIBFSYBDB/LIB,-
  SYBDB_OPTIONS/OPT, SYS$INPUT/OPT
    SYS$LIBRARY:VAXCRTL/SHARE
```

Summary



dberrhandle
dbmsghandle
Install Error handlers
dblogin( )
Get login structure
dbopen( )
Establish connection with SQL Server
FAIL
bad password etc.
dbcmd( )
dbfcmd( )
Set-up Query
Add SQL text to cmd buf
parsing compiling error
FAIL
dbsqlexec( )
1/batch
Submit Query to Server
dbresults( )
1/stmt
Get (Next) Set of Results
FAIL
permissions error
dbbind( )
Map Columns to Program Variables
dbnextrow( )
Get a Row
your code
Process Data

Lab Exercise: Environment Set-Up.

Lab Time: 20 minutes

The purpose of this lab is to have you set up your user account properly for the remainder of the course and build a skeleton program which can be used in the remainder of the labs.

All labs assume you have logged in to the operating system as userN (where N is indicated on your terminal) for Unix, and USERN for VMS. The password is the same as the login name.

1. Unix/C: copy **/usr/u/train/dblib/makefile** into your home directory.

   VMS/Fortran: Copy **SYBASE$SYSTEM:[SYBASE.TRAIN]DBLINK.COM** into your home directory.

2. Copy the errorhandlers into your directory as well. Unix: copy /usr/u/train/dblib/errorhandle.c. VMS: copy SYBASE$SYSTEM:[SYBSASE.TRAIN]errorhandle.for.

2. Create a program in your selected language (C or Fortran) which does a select * from sales and displays the rows on the screen, truncating the date to 12 characters (the date part only). Try and print the data in a columnar fashion. Be sure to include the errorhandlers.

3. Modify the LINK.COM or makefile to reference your program.

4. Compile, link and run your test program.

   Unix/C:
   1.) To compile and link: **make lab**
   2.) To run: **lab**

   VMS/Fortran:
   1.) To compile: **for/warn=dec <your program>**
   2.) To link: **@DBLINK.COM**
   3.) To run: **run <your program>**

Lab Answer

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

int err_handler();
int msg_handler();
main()
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        RETCODE  results;
        DBCHAR      title_id[7];
        DBCHAR  stor_id[5];
        DBCHAR  ord_num[21];
        DBCHAR  payterm[13];
        DBCHAR  dates[31];
        DBINT qty=1;
        BYTE    *answer;
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin();
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, " select * from sales");
        dbsqlexec(dbproc);
        while( dbresults(dbproc) != NO_MORE_RESULTS)
        {
        dbbind(dbproc, 1, NTBSTRINGBIND, -1, stor_id);
        dbbind(dbproc, 2, NTBSTRINGBIND, -1, ord_num);
        dbbind(dbproc, 3, NTBSTRINGBIND, -1, dates);
        dbbind(dbproc, 4, INTBIND, 0, &qty);
        dbbind(dbproc, 5, NTBSTRINGBIND, -1, payterm);
        dbbind(dbproc, 6, NTBSTRINGBIND, -1, title_id);
                while(dbnextrow(dbproc)!= NO_MORE_ROWS)
                {
                printf("%s\t %12s\t %11.11s\t %d\t %12s\t
%s\n",stor_id,ord_num,dates,qty,payterm,title_id);
                }
        }
        dbexit();
        exit();
}
```

SYBASE

Module 2

Row Buffering

Objective

- Access more than one row of results at a time (row buffering)

- Use Browse Mode routines to do updates in a multi–user environment

Row Buffering

- **Function**

  Allows programmed access to multiple results rows

  Allows dynamic access to any individual row in the buffer

  Each call to dbnextrow adds a row to the buffer until the buffer is full

- **Setting up row buffering**

  dbsetopt (dbproc, DBBUFFER, "no_of_rows")

  If no_of_rows is zero, default is 1000 rows

  Documented under "options"

- **Features**

  Independent of data access (dbbind and dbdata)

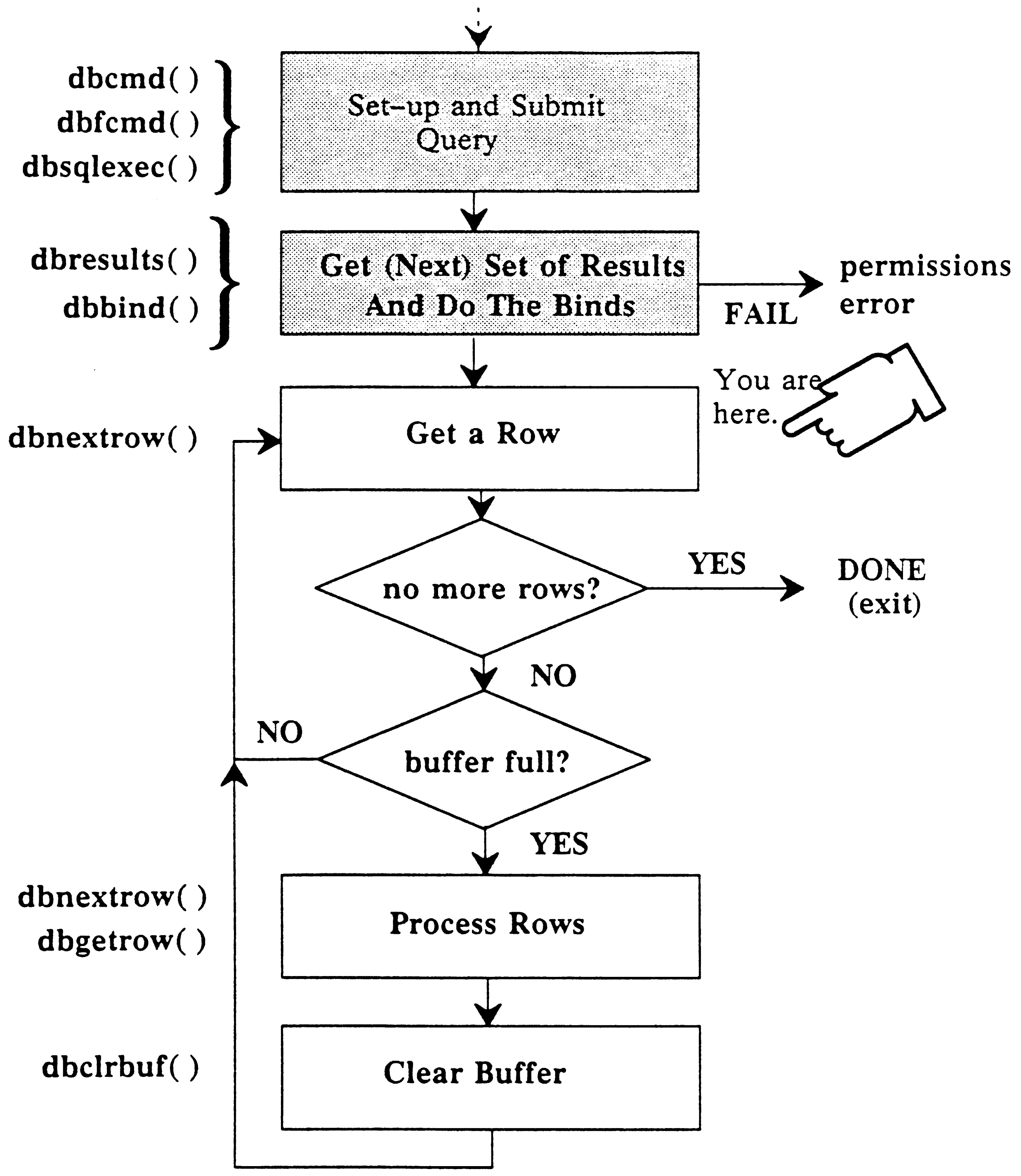
  Any row read from the server can be accessed in any order once the row is in the buffer

  Ability to find out the first, last and current rows

  Ability to clear "n" rows from the buffer to make room for more data

  Turn off row buffering for subsequent queries by using dbclropt(..)

Where Are We Now?



Row Buffering Functions

- **dbnextrow (dbproc)**

  Reads the next row from the SQL Server (normally)

  Binds any data to variables as needed

  Returns BUF_FULL if there is no more room in the buffer

  Returns NO_MORE_ROWS if there is no more data from the SQL Server

- **dbgetrow (dbproc, N)**

  Reposition pointers to Nth row of results. N is an integer.

  Note: "Nth" row means the Nth row coming back from the SQL Server, not the Nth row in the buffer

  Binds any data from row N to variables as needed

  Returns NO_MORE_ROWS if the row is not in the buffer

- **dbnextrow (dbproc) after dbgetrow**

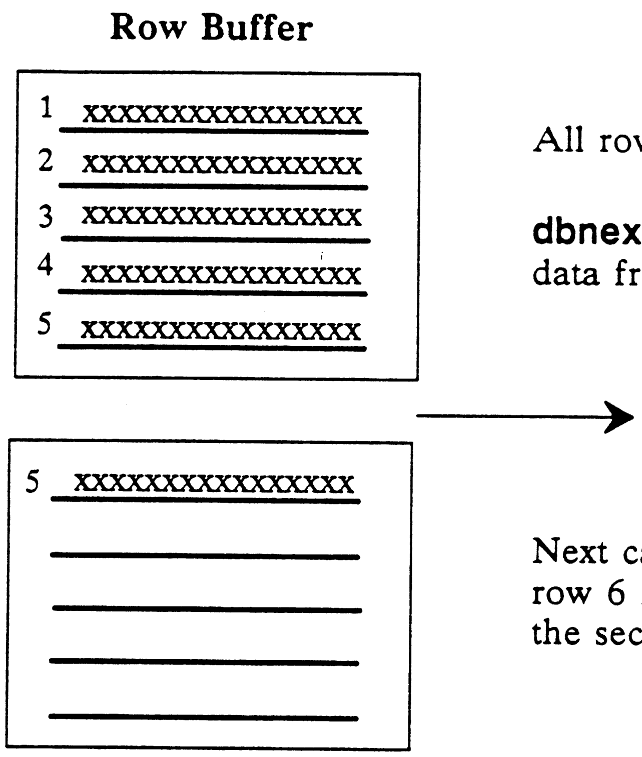
  Reads the next row directly from the buffer.

Row Buffering Functions (cont.)

- **dbclrbuf(dbproc, n)**

  Frees up n rows in the buffer by discarding oldest n rows

  If N = size of buffer, maximum of N–1 rows cleared



All rows filled

dbnextrow will not get more data from the Server

dbclrbuf(dbproc, 5)

Next call to **dbnextrow** puts row 6 from the Server into the second row of buffer

Row Buffering Functions (cont.)

- **DBLASTROW(dbproc)**
  DBFIRSTROW(dbproc)
  DBCURROW(dbproc)

  Return the appropriate SQL Server row number; useful in calls to dbgetrow

- **Example: get the first row in the buffer**

  C:

  ```
  dbgetrow(dbproc, DBFIRSTROW(dbproc));
  ```

  Fortran:

  ```
  row = fdbfirstrow(dbproc)
  call fdbgetrow(dbproc, row)
  ```

- **Example: get last row in buffer, then clear buffer**

  C:

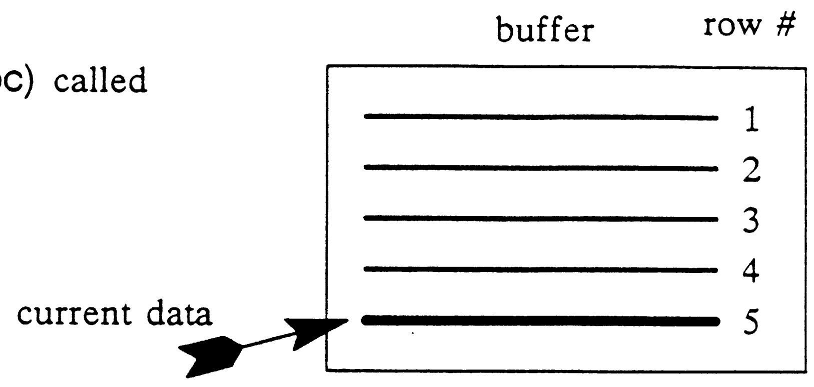
  ```
  dbgetrow(dbproc, DBLASTROW(dbproc));
  dbclrbuf(dbproc, N)
  ```

  Fortran:

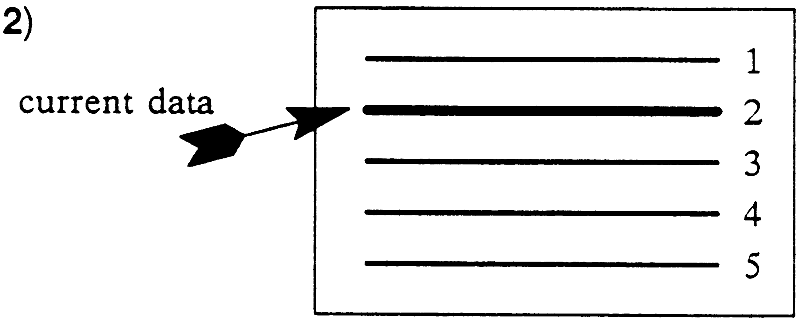
  ```
  row = fdblastrow(dbproc)
  call fdbgetrow(dbproc, row)
  call fdbclrbuf(dbproc, N)
  ```

Effects of dbnextrow, dbgetrow

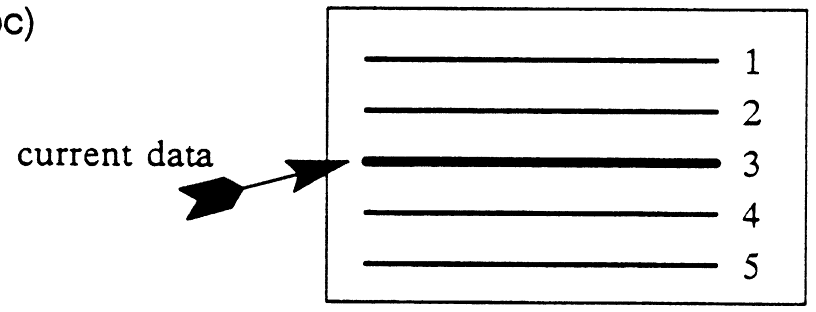
- **dbnextrow**(dbproc) called five times



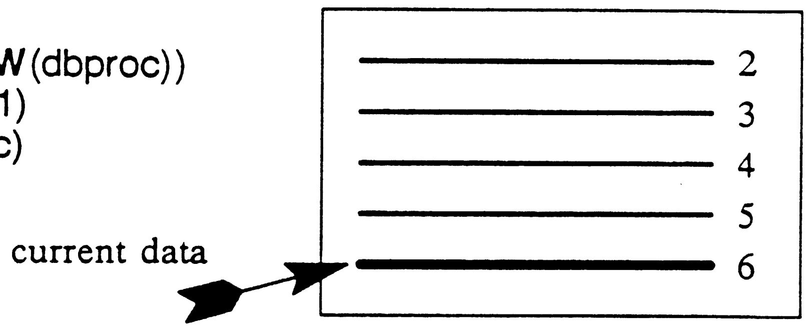
- **dbgetrow**(dbproc,**2**)



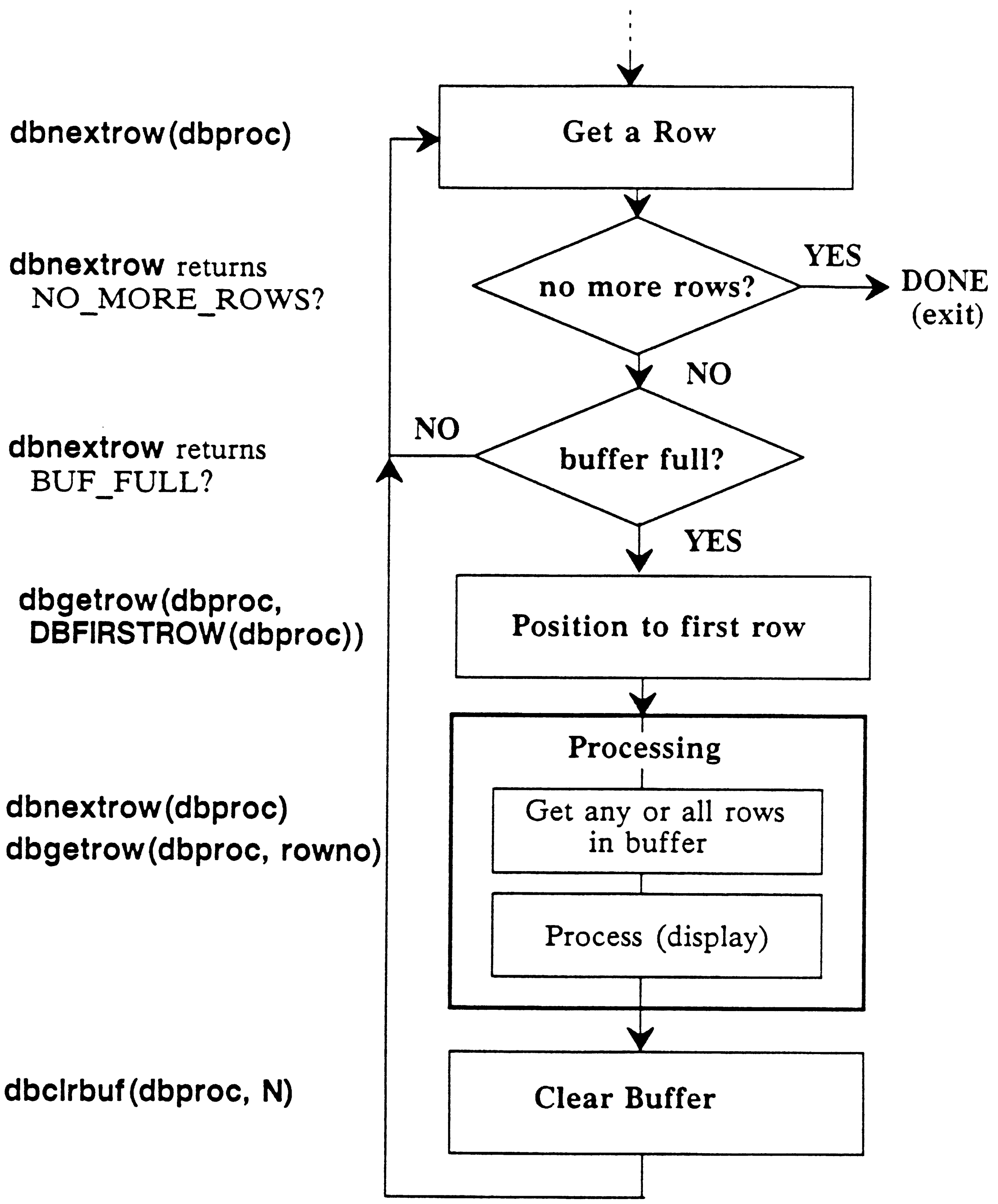
- **dbnextrow**(dbproc)



- **dbgetrow**(dbproc,
  DBLASTROW(dbproc))
  dbclrbuf(dbproc, 1)
  dbnextrow(dbproc)



Typical Usage



Example of Row Buffering

```
/* includes, error handlers omitted from this excerpt */
main( )
{
        DBPROCESS *dbproc;
        LOGINREC   *login;
        STATUS      results;
        int         row = 1, rownumber = 0;
        DBCHAR      title[81], price[9], title_id[10];
        login = dblogin();
        dbproc = dbopen(login, NULL);
        dbsetopt(dbproc, DBBUFFER, "5");
        dbcmd(dbproc, "select title, price, title_id from titles");
        if (dbsqlexec(dbproc) == FAIL))
        {
                printf("error in dbsqlexec\n");
                exit(ERREXIT);
        }
        dbresults(dbproc);
        dbbind(dbproc, 1, NTBSTRINGBIND, 0, title);
        dbbind(dbproc, 2, STRINGBIND, 0, price);
        dbbind(dbproc, 3, STRINGBIND, 0, title_id);
        while((results = dbnextrow(dbproc)) != NO_MORE_ROWS
                && results != BUF_FULL )
        {
                printf("%d %s\n", row++, title_id);
        }
        printf("Which row do you want? ");
        scanf("%d",&rownumber);
        if (dbgetrow(dbproc, rownumber) != NO_MORE_ROWS);
                printf("%d %s\n$%s\n", rownumber, title, price);
        else
                printf("Bad row number\n");
        dbexit( );
        exit(STDEXIT);
}
```

Fortran Row Buffering

```
C     include statement, error handlers omitted from this excerpt
       program  RowBuf
       INTEGER*4                dbproc, login, results, row, rownumber
       CHARACTER *(80)          title
       CHARACTER *(8)           price
       CHARACTER *(15)          title_id
       row = 1
       rownumber = 0
       login = fdblogin( )
       dbproc = fdbopen(login, NULL)
       call fdbsetopt(dbproc,DBBUFFER, '5')
       call fdbcmd(dbproc, 'select title, price, title_id from dbo.titles')
       if (fdbsqlexec(dbproc) .eq. FAIL) then
              type *, 'fdbsqlexec failed'
              call dbexit( )
              call exit
       end if
       call fdbresults(dbproc)
       call fdbbind(dbproc, 1, CHARBIND, 80, title)
       call fdbbind(dbproc, 2, CHARBIND, 0, price)
       call fdbbind(dbproc, 3, CHARBIND, 0, title_id)
       results = fdbnextrow(dbproc)
       do while ((results .ne. NO_MORE_ROWS) .and. (results .ne. BUF_FULL))
              row = row + 1
              type *, row, title_id
              results = fdbnextrow(dbproc)
       end do
       type *, ' Which row do you want to see?: (-1 to stop)'
       read *, rownumber
       if (fdbgetrow (dbproc, rownumber) .ne. NO_MORE_ROWS) then
              type *, rownumber, title, price
       else
              type *, 'Bad row number'
       end if
       call fdbexit( )
       call exit
       END
```

Browse & Update

- **To support browse and update**

  Display browse rows using row buffering

  Use a second DBPROCESS to do updates so that the results from the select remain untouched

- **SQL Precautions**

  Once you retrieve the data from the SQL Server, it can be changed in the data base; there is no guarantee that the update takes place on the same data as the select

- **SQL Browse**

  To guarantee the updated row hasn't changed, you must use the browse/timestamp feature

Browse Mode

- **Data Base Requirements**

  Table must include a column called "timestamp"

  ```
  create table test (a int, timestamp)   or ...
  alter table test add timestamp
  ```

  Table must have a unique index

- **Browse implies reading first, then updating only if the data has not changed**

  Select the data using the additional phrase "**for browse**"

  Update the data using a **where** clause referencing the timestamp column

  If the timestamp has changed, the update will fail

- **How to get and use the timestamp "where" clause**

  After a row has been retrieved (ie, after a dbnextrow):

  ```
  qualptr = dbqual (dbproc, -1, "tablename")
  ```

  When the update information is available:

  ```
  dbfcmd (dbproc, "update table set x=y %s",qualptr)
  ```

Browse Example

- Code Fragment

```
...
char  *qualptr;
...
dbcmd(dbproc, "select * from table for browse");
dbsqlexec(dbproc);
while (
       (return_code =dbresults(dbproc))
                   !=NO_MORE_RESULTS)
       /* do the dbbinds here   */
{
       if (return_code == SUCCEED)
       {
            while(dbnextrow(dbproc) != NO_MORE_ROWS)
             {
               qualptr = dbqual(dbproc, -1, "table");
               dbcmd (update_proc, "update table");
               dbfcmd(update_proc, " set x = 1 %s", qualptr);
              dbsqlexec(update_proc);
              dbresults(update_proc);
               free(qualptr);
             }
       }
}
```

- Notes

Use two dbprocs –one for select, one for update

If the update failed, you would have to do a select again

If you were using row buffering, you might have to do the select over yet a third dbproc

Other Browse Routines

- **Updating the same row twice**

  After the first update, the timestamp has changed.

  Use the following calls to set up the correct new timestamp:

  dbtsnewval(update_proc)

  returns the new ts value after the update is done

  dbtsnewlen(update_proc)

  returns the length of the new ts value

  dbtsput(dbproc, newval, newlen, -1, "table")

  updates the DB-Library information with the new timestamp so that subsequent calls to dbqual for this row will be correct

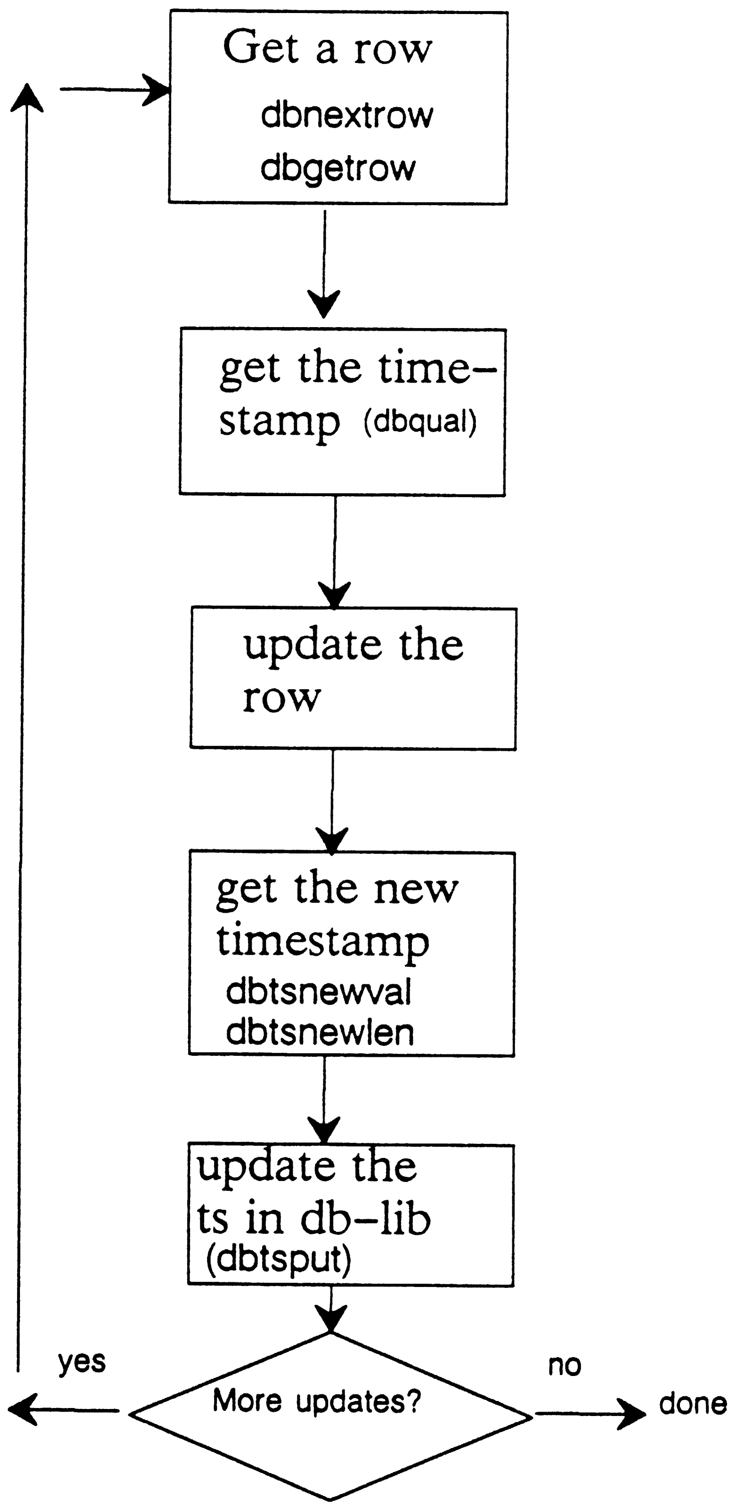
- **When to use these**

  Useful when using row buffering in conjunction with Browse

  Also useful if a particular row might be updated several times, each time changing a different column based on user input, for example

  Note dbtsput only updates the information that DB-Library has; the server has already made its changes to the timestamp during the update.

Sample Flow


Get a row
dbnextrow
dbgetrow
get the time-stamp (dbqual)
update the row
get the new timestamp
dbtsnewval
dbtsnewlen
update the ts in db-lib
(dbtsput)
More updates?
yes
no
done

Summary

- **Row Buffering allows multiple result rows to be accessed**

  dbsetopt(dbproc, DBBUFFER, "N") to turn it on

  dbnextrow to get next row from buffer or dataserver

  dbgetrow(dbproc,N) to get a row from the buffer

  dbclrbuf(dbproc,N) to remove rows from buffer

  DBLASTROW, DBFIRSTROW, DBCURROW to get row numbers

- **Browse and Update data**

  Use row buffering to see several rows

  Use a second DBPROCESS to do the updates

- **Browse Mode Routines and SQL syntax**

  select ... for browse

  dbqual to return pointer to timestamp and use the pointer in subsequent updates

  dbtsnewval, dbtsnewlen, dbtsput to keep timestamp accurate for re–updating same row

Lab Exercise: Row Buffering

Prior to beginning this lab, create your own copy of the titles table by going into isql and doing a select * into <your table name> from titles.

Pseudocode for these labs are on the following pages.

1. Write a program which allows the user to browse through your copy of the pubs titles table.

   - For each row in the table, display a row number, the title and the price, but retrieve the title_id as well.

   - After 5 rows are displayed, allow the user to type M for more rows, or X to exit.

   - If M is typed, display the next 4 rows, or however many are left in the database. When there are no more rows, exit the program.

   - If X is typed, exit the program.

2. In this exercise, we will assume there are NOT multiple users on the table, thus we can do the update with the assumption that the data has not changed since we last retrieved it. (ie, don't use browse mode).

   Modify the program above, and allow the user to also enter 'U' for "update." If U is typed,

   - Ask for a row number to update, and ask for the new price.

   - Update the row using the title_id for the selection, and setting the price to the new price (update titles set price = price where title_id = title_id).

   - Use a second DBPROCESS to do the update.

   - After the update, exit the program

(continued on next page)

Lab: Browse Mode.

3. Modify your copy of the titles table and add a unique index on the title_id field. Also add a timestamp column.

   Modify your update program in the previous example to use browse mode for the select and update.

Optional:

4. Modify the previous program to allow multiple updates by the user, including updating the same row. Test the program two ways: first comment out the code which maintains the in-core time stamp and notice what happens when you update the same row. Then add the proper code to maintain the timestamp validity, and re-run the program updating the same row.

Hints for C programmers:

Lab 1:

To get the user's choice, use scanf with a string format. Put the result in a char variable, not a string (char *) variable -- don't forget to send it as an address (&response, not response), since it's a single character, not a string.

To compare the inputted choice with a given value, use single quotes around a single character, ie.: if (response == 'M')...

Lab 2:

To read in the row number for the update, use scanf with an integer format, and store it in an integer variable. Again, don't forget to send this variable as an address (&rownumber) to scanf.

To read in the price, use scanf with a string format, and put it in a string variable; in this case, don't worry about using the ampersand -- strings automatically get sent as addresses in C, as do all arrays (a string is just a character array).

Pseudocode for labs

Lab 5.1

```
include statements, declare and install error handlers
set up DBPROCESS

turn on row buffering option

build and execute the select; if failed, error and exit
do a dbresults;

do the binds

set up dbnextrow loop:

        while there are still rows
                if the buffer isn't full
                        print out the rows
                else
                        get input from user
                        if 'X', exit
                        if 'M', clear buffer
                        else error, bad input
        end while
close and exit
```

Pseudocode, cont'd

Lab 5.2

```
include statements, declare and install error handlers
set up DBPROCESSes

turn on row buffering option

build and execute the select; if failed, error and exit
do a dbresults;

do the binds

set up dbnextrow loop:

    while there are still rows
        if the buffer isn't full
            print out the rows
        else
            get input from user
            if 'X', exit
            if 'U'
                get row number from user
                get row from buffer
                if bad row, print error, exit
                get new price
                build and execute update, inform user if failed
                exit
            if 'M', clear buffer
            else error, bad input
    end while

close and exit
```

Lab Answers

Problem 1

```
/* Lab Number 5.1 */
/* print out rows, ask user if he wants more or if he wants to quit */

/*  include statements, declare and install error handlers set up DBPROCESS */

#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

extern int err_handler();
extern int msg_handler();

main()
{
        DBPROCESS    *dbproc;
        LOGINREC     *login;
        STATUS               results; /* for dbnextrow returns */

        DBCHAR               title[81], price[9], title_id[10];
        int          row = 1;
        char         response;

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin();
        dbproc = dbopen (login, NULL);

        /* turn on row buffering option */
        dbsetopt(dbproc, DBBUFFER, "5");

        /* build and execute the select; if failed, error and exit */
        dbcmd(dbproc, "select title, price, title_id from titles");

        if(dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed");
                dbexit( );
                exit(ERREXIT);
        }

        /* do a dbresults */
        if (dbresults(dbproc) == FAIL)
                {
                printf("dbresults failed");
```

```
            dbexit( );
            exit(ERREXIT);
            }

    /* do the binds */
    dbbind(dbproc, 1, NTBSTRINGBIND, 0, title);
    dbbind(dbproc, 2, STRINGBIND, 0, price);
    dbbind(dbproc, 3, STRINGBIND, 0, title_id);

    /* set up dbnextrow loop */
    /* while there are still rows */
    while ((results = dbnextrow(dbproc)) != NO_MORE_ROWS)
    {
            /* if the buffer isn't full */
            if (results != BUF_FULL)
            {
                    /* print out the rows */
                    printf("%d  %s\n$%s\n",row++, title, price);
            }
            else
            {
                    /* get input from user */
                    printf("Please enter 'M' for more or 'X' to quit: ");
                    scanf("%s",&response);

                    /* if 'X', exit */
                    if (response == 'X')
                    {
                            printf("Goodbye\n");
                            dbexit();
                            exit(STDEXIT);
                    }

                    /* if 'M', clear buffer */
                    if (response == 'M')
                    {
                            dbgetrow(dbproc, DBLASTROW(dbproc));
                            dbclrbuf(dbproc, 5);
                    }

                    /* else error, bad input */
                    else
                            printf("Unknown  option\n");
            }               /* end else statement */
    }                       /* end dbnextrow while */

    /* close and exit*/
    dbexit();
    exit(STDEXIT);
}           /* end program */
```

Problem 2

```
/* Lab Number 5.2 */
/* this time, user can opt for updating; after an update, exit */
/* the program , or exit if no more rows */

/* include statements, declare and install error handlers */


#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

extern int err_handler();
extern int msg_handler();

main( )
{
        DBPROCESS    *dbproc1;
        DBPROCESS    *dbproc2;
        LOGINREC     *login;
        STATUS       results;
        DBCHAR       title[81], price[9], title_id[10], newprice[9];
        int          row = 1, rownumber;
        char         response;

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        /* set up DBPROCESSes */
        login = dblogin();
        dbproc1 = dbopen (login, NULL);
        dbproc2 = dbopen (login, NULL);

        /* turn on row buffering option */
        dbsetopt(dbproc1, DBBUFFER, "5");

        /* build and execute the select; if failed, error and exit */
        dbcmd(dbproc1, "select title, price, title_id from titles");
        if(dbsqlexec(dbproc1) == FAIL)
        {
                printf("dbsqlexec failed for select");
                dbexit();
                exit(ERREXIT);
        }

        /* do a dbresults */
        dbresults(dbproc1)

        /* do the binds */
        dbbind(dbproc1, 1, NTBSTRINGBIND, 0, title);
dbbind(dbproc1, 2, STRINGBIND, 0, price);
dbbind(dbproc1, 3, STRINGBIND, 0, title_id);

/* set up dbnextrow loop */
/* while there are still rows */
while ((results = dbnextrow(dbproc1)) != NO_MORE_ROWS)
{
        /* if the buffer isn't full */
        if (results != BUF_FULL)
                /* print out the rows */
                printf("%d  %s\n$%s\n",row++, title, price);
        else
        {
                /* get input from user */
                printf("Please enter 'M' for more, 'U' to update, ");
                printf("or 'X' to quit: ");
                scanf("%s",&response);

                /* if 'X', exit */
                if (response == 'X')
                {
                        printf("Goodbye\n");
                        dbexit();
                        exit(STDEXIT);
                }

                if (response == 'U')
                {
                        /* get row number from user */
                        printf("Please enter row number to update: ");
                        scanf("%d",&rownumber);

                        /* get row from buffer */
                        /* if bad row, print error, exit */
                        if(dbgetrow(dbproc1, rownumber) != REG_ROW)
                        {
                                printf("Bad row number");
                                dbexit( );   .
                                exit(ERREXIT);
                        }
                        / *get new price */
                        printf("Please enter the new price: ");
                        scanf("%s",newprice);

                        /* build and execute update, inform user if failed */
                        dbcmd(dbproc2, "update titles");
                        dbfcmd(dbproc2, " set price = $%s",newprice);
                        dbcmd(dbproc2, " where title_id");
                        dbfcmd(dbproc2, " = '%s'",title_id);

                        if (dbsqlexec(dbproc2) == FAIL)
                        {
```

```
                                        printf("update failed on dbsqlexec\n");
                                        dbexit();
                                        exit(ERREXIT);
                                }

                                /* exit */
                                printf("Update succeeded!\n");
                                dbexit();
                                exit(STDEXIT);
                        }

                        /* if 'M', clear buffer */
                        if (response == 'M')
                        {
                                dbgetrow(dbproc1, DBLASTROW(dbproc1));
                                dbclrbuf(dbproc1, 5);
                        }

                        /* else error, bad input */
                        else
                                printf("Unknown  option\n");

                }       /* end big else statement */
        }               /* end dbnextrow while */

        /* close and exit */
        dbexit();
        exit(STDEXIT);

}               /* end program */
```

Problem 3.

```
/* Lab Number 5.3 */
/* this time, use Broswe Mode */
/* same code as 5.2, except for lines in BOLD    */


/* include statements, declare and install error handlers */


#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

extern int err_handler();
extern int msg_handler();

main( )
{
        DBPROCESS    *dbproc1;
        DBPROCESS    *dbproc2;
        LOGINREC     *login;
        STATUS       results;
        DBCHAR       title[81], price[9], title_id[10], newprice[9];
        int          row = 1, rownumber;
        char         response;
        char         *qualptr;

        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        /* set up DBPROCESSes */
        login = dblogin();
        dbproc1 = dbopen (login, NULL);
        dbproc2 = dbopen (login, NULL);

        /* turn on row buffering option */
        dbsetopt(dbproc1, DBBUFFER, "5");

        /* build and execute the select; if failed, error and exit */
        dbcmd(dbproc1, "select title, price, title_id from titles");
        dbcmd(dbrpoc1, " for browse");
        if(dbsqlexec(dbproc1) == FAIL)
        {
                printf("dbsqlexec failed for select");
                dbexit();
                exit(ERREXIT);
        }

        /* do a dbresults */
        dbresults(dbproc1)
```

```
/* do the binds */
dbbind(dbproc1, 1, NTBSTRINGBIND, 0, title);
dbbind(dbproc1, 2, STRINGBIND, 0, price);
dbbind(dbproc1, 3, STRINGBIND, 0, title_id);

/* set up dbnextrow loop */
/* while there are still rows */
while ((results = dbnextrow(dbproc1)) != NO_MORE_ROWS)
{
        /* if the buffer isn't full */
        if (results != BUF_FULL)
                /* print out the rows */
                printf("%d  %s\n$%s\n",row++, title, price);
        else
        {
                /* get input from user */
                printf("Please enter 'M' for more, 'U' to update, ");
                printf("or 'X' to quit: ");
                scanf("%s",&response);

                /* if 'X', exit */
                if (response == 'X')
                {
                        printf("Goodbye\n");
                        dbexit();
                        exit(STDEXIT);
                }

                if (response == 'U')
                {
                        /* get row number from user */
                        printf("Please enter row number to update: ");
                        scanf("%d",&rownumber);

                        /* get row from buffer */
                        /* if bad row, print error, exit */
                        if(dbgetrow(dbproc1, rownumber) != REG_ROW)
                        {
                                printf("Bad row number");
                                dbexit( );
                                exit(ERREXIT);
                        }
                        qualptr = dbqual(dbproc1, -1, "titles");
                        / *get new price */
                        printf("Please enter the new price: ");
                        scanf("%s",newprice);

                        /* build and execute update, inform user if failed */
                        dbcmd(dbproc2, "update titles");
                        dbfcmd(dbproc2, " set price = $%s",newprice);
                        dbfcmd(dbproc2, "  %s ", qualptr);
                        if (dbsqlexec(dbproc2) == FAIL)
                        {
                                printf("update failed on dbsqlexec\n");
                                dbexit();
                                exit(ERREXIT);
                        }

                        /* exit */
                        printf("Update succeeded!\n");
                        dbexit();
                        exit(STDEXIT);
                }

                /* if 'M', clear buffer */
                if (response == 'M')
                {
                        dbgetrow(dbproc1, DBLASTROW(dbproc1));
                        dbclrbuf(dbproc1, 5);
                }

                /* else error, bad input */
                else
                        printf("Unknown  option\n");

        }       /* end big else statement */
    }           /* end dbnextrow while */

    /* close and exit */
    dbexit();
    exit(STDEXIT);

}           /* end program */
```

Problem 4:

```
/* Lab Number 5 */
/* a differnt version of browse, this time it loops back for more updates, and */
/*  keeps the in-core pointer up to date for the timestamp  */

#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

extern int err_handler();
extern int msg_handler();

main()
{
  DBPROCESS  *dbproc1;
  DBPROCESS  *dbproc2;
  LOGINREC   *login;
  RETCODE    results;
  DBCHAR     title[81],price[9], title_id[10], newprice[9];
  int        row = 1, rownumber = 0;
  char       response;
  char       *qualptr;

  dbmsghandle(msg_handler);
  dberrhandle(err_handler);

  login = dblogin();
  dbproc1 = dbopen (login, NULL);
  dbproc2 = dbopen (login, NULL);

  dbsetopt(dbproc1, DBBUFFER, "5");

  dbcmd(dbproc1, "select title, price, title_id from titles");
  dbcmd(dbproc1, " for browse");
  dbsqlexec(dbproc1);

  while ((results = dbresults(dbproc1)) != NO_MORE_RESULTS)
  {
       if (results != FAIL)
       {
         dbbind(dbproc1, 1, NTBSTRINGBIND, 0, title);
         dbbind(dbproc1, 2, STRINGBIND, 0, price);
         dbbind(dbproc1, 3, STRINGBIND, 0, title_id);

         while ((results = dbnextrow(dbproc1)) != NO_MORE_ROWS)
         {
              if (results != BUF_FULL)
              {
                printf("%d  %s\n$%s\n",row++, title, price);
              }
```

```
            else
            {
             printf("Please enter 'M' for more or 'U' to update: ");
             scanf("%s",&response);

             while (response != 'M')
             {
                    printf("Please enter row number to update: ");
                    scanf("%d",&rownumber);

                    /* check for returns on dbgetrow */
                    if ((results = dbgetrow(dbproc1, rownumber)) !=
                       REG_ROW)
                    {
                      if (results == NO_MORE_ROWS)
                            printf("Row out of bounds\n");
                      else printf("That is a compute row, sorry\n");
                    }
                    else
                    /* it was a REG_ROW */
                    {
                    qualptr = dbqual(dbproc1,-1, "titles");
                      printf("Please enter the new price: ");
                      scanf("%s",newprice);

                      dbcmd(dbproc2, "update titles");
                      dbfcmd(dbproc2, " set price = $%s",newprice);
                      dbfcmd(dbproc2, " %s", qualptr);
                      if ((results = dbsqlexec(dbproc2)) == SUCCEED)
                      {
                            printf("Update successful\n");
                           dbtsput(dbproc1, (dbtsnewval(dbproc2)),
                                   (dbtsnewlen(dbproc2)),-1, "titles");

                      }
                      else printf("Update unsuccessful\n");
                    }

                   /* input for another command */
                   printf("Please enter 'M' for more or 'U' to update: ");
                   scanf("%s",&response);

             }              /* end user command while loop */

             /* if exited this loop, user pressed 'M' */
             dbgetrow(dbproc1, DBLASTROW(dbproc1));
             dbclrbuf(dbproc1, 5);

           }                /* end else statement */
      }                     /* end dbnextrow while */
     }                      /* end if (results != FAIL) */
}                           /* end dbresults while */
```

```
    /* close the processes */
    dbclose(dbproc1);
    dbclose(dbproc2);
    dbexit();

}                   /* end program */
```

SYBASE

Module 3

Text Data

Stored Procedures

Objectives

- Use DB–Library to send text or image data to the SQL Server

- Use the Remote Procedure Protocol to send stored procedures to the SQL server

- Use DB–Library calls to process return values and return parameters from procedures

Text & Image Handling

- **Text/Image Datatypes**

  Can hold up to 2 gigabytes of data

  Can be added or included in a table definition

  Stored in the database in its own data pages, with a minimum size of 1 page

  Use only if needed since it requires additional disk access as well as substantial space requirements

- **Accessing Text/Image through SQL**

  Insert, Update are limited to 128K of data

  Select and Delete default to 32K of data, or the value in "set textsize n" or dbsetopt(dbtextlimit)

  Portions of the column can be read using readtext

  Text/image columns can be updated (completely replaced) using writetext

- **Writing more than 128K of data using DB-Library**

  dbwritetext( ) – updates a text value more efficiently than SQL writetext

  dbmoretext( )– allows you to send portions of the data at a time

Reading and Writing Text Data

- The *readtext* command retrieves part or all of a text data item
- *readtext* uses the text pointer, an offset and a length to read any part of the text data item
- Use *textptr* to get a text pointer

```
declare @pointer_name varbinary
select @pointer_name = textptr(col_name)
from table_name
where search_conditions
```

- Use *readtext* to read the text data

```
readtext table.column text_pointer offset size
```

Example:

Suppose we have a database called reports, which holds research papers. The text of each paper is in a *text* column called **article**. We want to read the first 2K of the paper written by Goldschmidt in 1987.

```
declare @txtptr  varbinary
select @txtptr = textptr(article)
from reports
where au_lname = 'Goldschmidt'
and pub_year = 1987

readtext reports.article @txtptr  0  2000
```

- Use **writetext** to update a text data item

```
writetext table.column text_pointer [with log] data
```

Initializing Text Fields

- **How it works**

  The initial state of a null text column (ie., after create table) is that the data page is not allocated, and there is a NULL text pointer in the row

  If you insert null text data the data page is still unallocated, to avoid wasting disk space

  However, writetext and dbwritetext require a valid (non-null) text pointer.

  Thus when there is null text data, you must precede writetext or dbwritetext with an update, supplying null or data, in order to establish a valid text pointer

- **Example:** create table myt (a int, b text NULL)

  This will fail due to Null text pointer:

```
insert table  myt (a) values (1, NULL)
declare @textptr varbinary(16)
select @textptr = textptr(b) from myt where a = 1
writetext myt.b @textptr with log "new data"
```

  This will work:

```
update myt set b = NULL where a = 1
declare @textptr varbinary(16)
select @textptr = textptr(b) from myt where a = 1
writetext myt.b @textptr with log "new data"
```

Using DB-Library

- **Basic Text/image functions**

  dbtxptr(dbproc, column) – get a pointer

  dbtxtimestamp(dbproc,column) – get the timestamp

  dbwritetext(...) – update the text data using the above information to identify the row

- **Additional functions**

  If you need to write to the same row twice:

  dbtxttsnewval( ) – get the new timestamp value
  dbtxtsput( )– put the new timestamp value in the DBPROC for future retrieval

- **Advantages of using dbwritetext**

  Allows the data to be inserted/updated without logging

  Allows more than 128k of data to be sent to the Server

  More efficient use of memory and better performance than using insert

Using dbwritetext

- **Simple mode – send all the text at once**

Parameters:

dbproc:	A dbproc to use for the update; if results are pending from a select, use a second dbproc for writetext
Object name:	"Table.Column"
Text Ptr:	Results of calling dbtxptr
Length:	DBTXPLEN (defined in header files)
Timestamp:	Results of calling dbtxtimestamp
Log:	TRUE – log update FALSE – requires dboption "select into"
Size	Amount of text being written
Text	Pointer to the text to transfer to server

- **Example**

```
dbwritetext(
   dbproc2, "titles.title", dbtxptr(dbproc,1), DBTXPLEN,
   dbtxtimestamp(dbproc,1), TRUE, 20, mytext );
```

- **Sending data piecemeal**

Change Text to NULL in dbwritetext; then for each portion of the text, use dbmoretext

See the documentation for more information

Simple Example

```
(put include files, etc. here first)
int err_handler();
int msg_handler();
main()
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        DBCHAR mytext[200];
        DBCHAR title_id[10];
        DBBINARY  *txptr;
        DBBINARY  *times;
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin();
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, "select mytext,title_id from titles");
        dbcmd(dbproc, " where title_id = 'BU1032'");
        dbsqlexec(dbproc);
        while (dbresults(dbproc) != NO_MORE_RESULTS)
        {
        dbbind(dbproc,1,STRINGBIND,0,mytext);
        dbbind(dbproc,2,STRINGBIND,0, title_id);
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
                {
                printf ("  %s\n %s\n ",title_id,mytext );
                txptr = dbtxptr(dbproc, 1);
                times = dbtxtimestamp(dbproc, 1);
                printf (" \n type in new text:\n");
                gets (mytext);
                dbwritetext(dbproc, "titles.mytext", txptr,
                        DBTXPLEN, times, TRUE, 10, mytext);
                }
        }
```

Try it now!

Modify your copy of the titles table and add a text column (allowing nulls) called "review" to the table. Initial value of the text column will be null.

Write a DB-Library program which gets a title-id from the user, and displays the title. Prompt the user for the text for the "review" column, and put that review in the database.

You can use a modified version of lab answer 4.1 if you still have it on-line. You can test the program initially by simply using update statements as in the earlier lab. However the program should be ultimately written and running using a call to **dbwritetext.**

Reminder: you cannot use **dbwritetext** if the data is currently null. Update the data to non-null values before using the program.

Optional Lab:

Modify your previous Browse Mode program to allow the user to input a new Review column instead of a new price. Assume the user will only update a given row once during the run (ie, don't bother with the text timestamp). Again, use **dbwritetext** to make the change to the database.

Lab Answer - C

```
/* Lab Number  */
/* prompt for title_id; if valid, retrieve and print title of */
/* corresponding book, ask for review update the text field review */

#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

extern int err_handler();
extern int msg_handler();

main()
{

        DBPROCESS       *dbproc;
        LOGINREC        *login;
        RETCODE         return_code;

        DBCHAR          title_id[7]; /* one longer than tid length */
        DBCHAR          title[81];  /* one longer than title max length */
        DBCHAR          review[101];

        /* install the error handlers */
        dbmsghandle(msg_handler);
        dberrhandle(err_handler);

        login = dblogin();

        dbproc = dbopen(login,NULL);
        if (dbproc == NULL)
        {
                printf("error in dbproc\n");
                exit();
        }

        /* prompt the user */
        printf("Please enter a title-id: ");
        scanf("%s",title_id);
        dbcmd(dbproc," select title, review from mytitles");
        dbfcmd(dbproc," where title_id = '%s'", title_id);

        if (dbsqlexec(dbproc) == FAIL)
          {
                printf("error in dbsqlexec\n");
                exit();
        }

        while ( (return_code = dbresults(dbproc)) != NO_MORE_RESULTS)
        {
                if (return_code == FAIL)
                        printf("Fail in dbresults\n");

                else if (DBROWS(dbproc) != SUCCEED)
                {
                        printf("Invalid title id\n");
                        printf("Please enter a title id: ");
                        scanf("%s",title_id);

                        dbcmd(dbproc," select title, review from mytitles");
                        dbfcmd(dbproc," where title_id = '%s'", title_id);

                        if (dbsqlexec(dbproc) == FAIL)
                        {
                                printf("error in dbsqlexec\n");
                                exit();
                        }
                }

                else
                {
                        dbbind(dbproc,1,STRINGBIND,0,title);
                        dbbind(dbproc,2,STRINGBIND,0, review);
                        while(dbnextrow(dbproc) != NO_MORE_ROWS)
                        {
                                printf("Title: %s\n",title);
                                printf("review: %s\n", review);

                        printf("Please enter Book Review: ");
                        scanf("%[^/]",review);
                        dbwritetext(dbproc, "mytitles.review", dbtxptr(dbproc,2)
                        DBTXPLEN, dbtxtimestamp(dbproc,2),TRUE, 100,review);
                        }
                }
        }
dbclose(dbproc);
dbexit();
}
```

dbsqlexec is intern : → dbsqlsend

[Doe wait]

if (dbSQL ok ..)

Using RPC

- **What is it?**

  Performance enhanced alternative to dbsqlexec for running stored procedures

  Bypasses parse and compile of the SQL exec statement

  Allows parameters to be sent in native, rather than ascii format

  Automatically used by the Server itself when executing stored procedures on other servers

- **Programming components**

  dbrpcinit – initializes the call, includes the stored procedure name as a parameter

  dbrpcparam – adds a parameter to the execution stream for this rpc call

  dbrpcsend – sends the whole stream to the server

  dbsqlok – wait for results

- **After dbsqlok returns**

  Normal calls (dbresults, dbnextrow, etc) can be used.

Bij sp_recompile <table> groeit de procedure steeds iets

na een tijd: procedure droppen en opnieuw creëren

Simple Programming

- **dbrpcinit (dbproc,** *procname,* **0)**

  Initializes an rpc stream to execute the procedure

  procname can be a pointer to the procedure name itself, or you can put the name right in the call, ie., dbrpcinit (dbproc, "myproc", 0)

  The last parameter is 0, or DBRPCRECOMPILE to force recompilation of the stored procedure

- **dbrpcsend(dbproc)**

  Cause the execution to take place

- **Returns from dbrpcinit, dbrpcsend**

  SUCCEED or FAIL

- **Sample code**

```
dbrpcinit(dbproc, "myproc", 0);
dbrpcsend(dbproc);
dbsqlok(dbproc);
while (dbresults(dbproc) != NO_MORE_RESULTS)
{
  do binds here
  while (dbnextrow(dbproc) != NO_MORE_ROWS)
          {
          do processing here
          }
```

Passing Parameters

- **dbrpcparam (dbproc, ...)**

  Called once for each parameter to the stored procedure

  Must be issued prior to dbrpcsend

  Supports passing parameters by name (specify name with @name) or by order (omit names and the order of the calls to dbrpcparam determines the parameter order)

- **Parameters for dbrpcparam**

  Parameter name: pointer to ascii name of parameter, ie, "@param1" or NULL if passing by value

  Status: 0 or DBRPCRETURN (for return parameters)

  Type: Datatype as defined in stored procedure (SYBINT1, SYBCHAR, etc.)

  Maxlen: −1 or max length for variable length return parameters

  Datalength: −1 or actual length of data (not including null terminator) for parameters whose type is variable length (char, text, etc.). 0 indicates a null parameter value

  Value: pointer to the parameter data itself

Parameter Examples

```
create procedure myproc @type varchar(15)
as  select title_id from titles where type = @type

       DBCHAR  type[15];
       ...
       dbrpcinit(dbproc, "myproc", 0);
       strcpy(type, "business");
       dbrpcparam(dbproc, "@type", 0, SYBCHAR, -1, 8, type);
       dbrpcsend(dbproc);
       dbsqlok(dbproc);
       while (dbresults(dbproc) != NO_MORE_RESULTS)
       {
         dbbind(dbproc,1,STRINGBIND,0, title_id);
         while (dbnextrow(dbproc) != NO_MORE_ROWS)
       etc...

create procedure myp @flag int = 0 as
if @flag = 0
select title_id from titles where type = "business"
else
select title_id from titles where type = "psychology"

       DBINT       flag = 1;
       ...
       dbrpcinit(dbproc, "myp", 0);
       dbrpcparam(dbproc, "@flag", 0, SYBINT4, -1, 8, &flag);
       dbrpcsend(dbproc);
       dbsqlok(dbproc);
       while (dbresults(dbproc) != NO_MORE_RESULTS)
       {
         dbbind(dbproc,1,STRINGBIND,0, title_id);
         while (dbnextrow(dbproc) != NO_MORE_ROWS)
       etc...
```

kan ook -1 zijn : default

Try It Now

Write a DB–Library program which uses the RPC protocol to execute a stored procedure and display results. The stored procedure should do a select from titles based on type where type is a parameter passed to the stored procedure.

Easy method: hard code the contents of the "type" parameter into the application

Harder method: prompt the user for the "type", and pass that value as the parameter.

Lab Answer

```
/*  create procedure myproc (@type varchar(15) ) */
/*  as  select title_id from titles where type = @type    */

/*  put include files here...   */

int err_handler();
int msg_handler();
main()
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        DBCHAR   title_id[7];
        DBCHAR  type[15];
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin();
        dbproc = dbopen (login, NULL);
        dbrpcinit(dbproc, "myproc", 0);
        strcpy(type, "business");
        dbrpcparam(dbproc, "@type", 0, SYBCHAR, -1, 8, type)
        dbrpcsend(dbproc);
        dbsqlok(dbproc);
        while (dbresults(dbproc) != NO_MORE_RESULTS)
        {
           dbbind(dbproc,1,STRINGBIND,0, title_id);
           while (dbnextrow(dbproc) != NO_MORE_ROWS)
                        {
                        printf ("   next row  %s \n", title_id );
                        }
           }
        dbexit();
        exit();
}
```

SQL Return Parameters

- **Defining Return Parameters**

  Parameters can be defined as return parameters, using the keyword *output*

  The value of return parameters can be passed back to the caller

  Both the caller and the procedure must declare the parameter as *output* for the caller to receive the returned value

- **SQL Example:**

```
/*    Procedure to return the number of sales for a title */

create proc num_sales
   (@tid           char(6),
    @total_sales int        output )
as
select @total_sales = sum(qty)
   from sales
      where title_id = @tid
return

declare @total_sales int
exec num_sales 'PS2091', @total_sales output
```

- **DB–Library considerations**

  If using RPC protocol, use DBRPCRETURN for the status field in dbrpcparam . This is the equivalent of using "output" in the SQL exec statement

Programming for Return Parameters

- **Obtaining values using the DB–Library Routines**

  Use these regardless of whether you are using RPC or dbsqlexec to run the stored procedures

  Can only be called after all results and rows have been processed (after dbresults/dbnextrow is complete)

  Return values are sequentially numbered in the order in which they are specified in the create procedure statement, not counting non–return parameters

- **dbretdata(dbproc, *n*)**

  Returns a byte pointer to the data for return parameter *n*

- **Miscellaneous routines useful for ad–hoc or general purpose programs**

dbnumrets(dbproc) –	# of return values generated
dbretlen(dbproc, *n*) –	returns the integer length of the *n*th return parameter
dbrettype(dbproc, *n*) –	returns data type (for use with dbprtype)
dbretname(dbproc, *n*) –	returns name of *n*th return parameter

Example

```
create procedure myproc (@type varchar(15), @howmany int out)
 as select title_id from titles where type = @type
 select @howmany = @@rowcount
```

- **SQL code**

```
1> declare @amt int
2> exec myproc "business", @howmany = @amt output
3> go
 title_id
 BU1032
 BU1111
 BU2075
 BU7832
Return parameters:
@howmany
        4
```

- **DB–Library excerpt**

```
BYTE   *answer;
dbrpcparam(dbproc, "@type", 0, SYBCHAR, -1, 8, "business");
dbrpcparam(dbproc, "@howmany", DBRPCRETURN, SYBINT4, -1,
                                      -1, &howmany);
dbrpcsend(dbproc);
dbsqlok(dbproc);
while (dbresults(dbproc) != NO_MORE_RESULTS)
        {
         dbbind(dbproc,1,STRINGBIND,0, title_id);
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
                {
                printf ("   next row  %s \n", title_id );
                }
        }
        answer = dbretdata(dbproc,1);
        printf (" return value from proc: %d \n ", *(DBINT *)answer);
```

Try it Now!

Create a program which will run the following stored procedure and print out the results. The parameter "type" can be hard-coded in the program (ie, strcpy(type, "business"), etc.)

```
create procedure myproc (@type varchar(15), @howmuch char out)
as
select @howmuch = max(price) from titles where type = @type
```

Lab Answer

```
/ * create procedure myproc (@type varchar(15), @howmuch char(15)
out ) */
/* as select @howmuch = max(price) from titles where type = @type */

#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

int err_handler();
int msg_handler();
main()
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        DBCHAR   howmuch[15];
        BYTE   *answer;
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin();
        dbproc = dbopen (login, NULL);
        dbrpcinit(dbproc, "myproc", 0);
        dbrpcparam(dbproc, "@type", 0, SYBCHAR, -1, 8,"business");
        dbrpcparam(dbproc, "@howmuch", DBRPCRETURN, SYBCHAR,
15, 0, howmuch);
        dbrpcsend(dbproc);
        dbsqlok(dbproc);
        dbresults(dbproc);
        printf(" here ok \n ");
        answer = dbretdata(dbproc,1);
        printf (" return value from proc: %s \n ", *(DBCHAR *)answer);
        dbexit();
        exit();
}
```

Returning Procedure Status

- **Every procedure automatically returns a return status**

  Return status is always an integer value

  Zero and –1 through –99 return status is reserved for use by Sybase; (see Command Reference Manual)

  You can use positive or <–99 return status numbers to inform the caller of user–defined conditions

- **You can test for the return status or ignore it**

  Example: defining a procedure to return status

```
/*    Procedure to return the contract status for an author
**    Returns -900 if the author does not exist */

create proc contract
   (@name  varchar(40) )
as
if not exists (select * from authors
               where au_lname = @name)
   return -900       /* error status */

select au_id, contract, au_lname
from authors
where contract = 1
and au_lname = @name
return 0
```

- **DB–Library Calls**

  dbretstatus(dbproc) – returns the status

  dbhasretstat(dbproc) – useful for ad hoc queries.

Summary

- **Programming with Stored Procedures**

  Use RPC when possible for performance

  dbrpcinit
  dbrpcparam
  dbrpcsend
  dbsqlok

  Use return status to indicate results of the stored procedures

  dbretstatus

  In general, put as much code as possible in stored procedures, including data conversion and returning values to programs

  dbretdata

- **Programming with Text Data**

  Use text only when necessary

  Use dbwritetext to insert or change more than 128k of data

Exercise: Application

1. Write a stored procedure to print the store id, title id, and quantity of all the sales since a given date. (The date will be an input parameter). Return a status of 999 of there are no sales. Also, return the maximum number of all the sales for that period to the calling program.

   Test the stored procedure using ISQL.

2. Write a DB-Library program which runs the stored procedure using RPC protocol. Print the results and the maximum number of sales.

Optional: Print out a message if the return status is 999.

Lab Answer

```
/* create proc get_sales (@cutoff char(8), @max int output) */
/* as */
/* if not exists */
/*          (select * from sales */
/*          where date > @cutoff) */
/*          return -999 */
/* */
/* select stor_id, title_id, qty from sales */
/* where date > @cutoff */
/* */
/* select @max = max(qty) */
/* from sales */
/* where date > @cutoff */
/* */
/* return 0 */

#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

int err_handler();
int msg_handler();
main()
{
          DBPROCESS *dbproc;
          LOGINREC *login;
          RETCODE  results;
          DBCHAR  title_id[9];
          DBCHAR  stor_id[7];
          DBCHAR  dates[15];
          DBINT     qty=1;
          BYTE    *answer;
          dberrhandle(err_handler);
          dbmsghandle(msg_handler);
          login = dblogin();
          dbproc = dbopen (login, NULL);
          printf(" Type in date to begin the sales listing from: \n");
```

```
        gets(dates);
        printf ( "Looking for date:  %s \n", dates);

        dbrpcinit(dbproc, "get_sales", 0);
        dbrpcparam(dbproc, NULL, 0, SYBCHAR, -1, strlen(dates),dates);
        dbrpcparam(dbproc, NULL, DBRPCRETURN, SYBINT4, -1, 0, &qty);
        dbrpcsend(dbproc);
        dbsqlok(dbproc);

        while(     dbresults(dbproc) != NO_MORE_RESULTS)
        {
        dbbind(dbproc, 1, STRINGBIND, 0, stor_id);
        dbbind(dbproc, 2, STRINGBIND, 0, title_id);
        dbbind(dbproc, 3, INTBIND, 0 ,&qty);
                while(dbnextrow(dbproc)!= NO_MORE_ROWS)
                {
                printf(" %s %s %d  \n", stor_id, title_id, qty, dates);
                }
    }
        answer = dbretdata(dbproc,1);
        printf (" return value from proc: %d \n ", *(DBINT *)answer);
        dbexit();
        exit();
}
/*  */
```

SYBASE

Module 4

Miscellaneous

Topics

- Handling compute data
- Programming for Ad Hoc queries
- Access results data without having to allocate a variable for each column returning (**dbdata**). Good for *ad hoc* queries
- Use conversion routines to change data types (**dbconvert**)
- Programming Techniques
- Discussion of individual applications

Handling Compute Data

- **SQL example of compute**

```
select type, price from titles
order by type, price
compute sum(price) by type
```

- **Function**

  Provide summary data in a separate data row as opposed to adding a column (as aggregates normally do)

- **DB-Library functions**

  dbnextrow returns a compute id when the row is a compute row, and REG_ROW when the row is selected data

  dbaltbind allows you to bind compute data to variables

  dbadata allows you to access compute data

- **Program considerations**

  If you know the select statement had a compute clause, be sure to check the return from dbnextrow for REG_ROW, NO_MORE_ROWS, BUF_FULL. If it is none of these, then the data is a compute row

Compute Example – C

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
extern int err_handler( );
extern int msg_handler(  );
main( )
{
      DBPROCESS *dbproc;
      LOGINREC *login;
      DBINT       interim;
      STATUS  results;
      DBFLT8     sum_price;
      dberrhandle(err_handler);
      dbmsghandle(msg_handler);
      login = dblogin( );
      dbproc = dbopen (login, NULL);
      dbcmd(dbproc, "select type, price from titles");
      dbcmd(dbproc, " order by type, price");
      dbcmd(dbproc, " compute sum(price) by type");
      if (dbsqlexec(dbproc) == FAIL)
      {
            printf("dbsqlexec failed\n");
            dbexit( );
            exit(ERREXIT);
      }
      dbresults(dbproc);
      dbaltbind (dbproc, 1,1, FLT8BIND,0,&sum_price);
      while ((results = dbnextrow(dbproc)) != NO_MORE_ROWS)
      {
            if (results != REG_ROW)
                  printf ("Totals:  %.2f\n", sum_price);
      }
      dbexit( );
      exit(STDEXIT);
}
```

Compute Example – Fortran

```
program PrintComputes
include '(fsybdb)'
INTEGER*4       dbproc
INTEGER*4       login
INTEGER*4       results
REAL*8     price
EXTERNAL err_handler
EXTERNAL msg_handler
call fdberrhandle(err_handler)
call fdbmsghandle(msg_handler)
login = fdblogin()
dbproc = fdbopen(login,NULL)
call fdbcmd(dbproc, 'select type, price from dbo.titles')
call fdbcmd(dbproc, '  order by type, price')
call fdbcmd(dbproc, ' compute sum(price) by type')
if (fdbsqlexec(dbproc) .eq. FAIL) then
      type *, 'dbsqlexec failed'
      call fdbexit
      call exit
end if
call fdbresults(dbproc)
call fdbaltbind(dbproc,1,1,FLT8BIND,0,price)
results = fdbnextrow(dbproc)
do while (results .ne. NO_MORE_ROWS)
      if (results .ne. REG_ROW) then
            type *,'Total:  ',price
      end if
      results = fdbnextrow(dbproc)
end do
call fdbexit()
call exit
END
```

Ad Hoc Queries

- **Definition**

  Your program doesn't know the contents of the command buffer; thus it doesn't know what results to expect

- **Examples**

  Prompting the user to type in a select which you simply pass to the server

  Writing a front end program such as isql

- **Programming**

  Use functions after dbresults to tell you how much and what kind of data was returned:

DBROWS	(were any rows returned?)
DBCMDROW	(was the command a select?)
DBCOUNT	(returns no. of rows affected by a SQL command)

  Use functions after dbresults to get Meta Data about the rows

  Column count

  Column names, etc

  Use dbdata, dbdatlen

New Functions

- **dbtabcount (dbproc)**

  returns number of tables used in this query

- **dbtabname(dbproc, tabnum)**

  returns a pointer to the null–terminated name of the table

- **dbtabsource(dbproc, colnum, tabnum)**

  returns name and optionally number of the table from which this *nth* column was derived

- **dbcolsource(dbproc, colnum)**

  returns the name of the actual database column, regardless of the "header" specified in the select (for *select author = au_id..* colsource returns au_id)

  returns NULL if the column is the result of an expression

- **dbcolbrowse**
  dbtabbrowse

  indicates if the column or table can be updated using browse mode

DB-Library functions for ad-hoc queries

Query:
```
select * from sales order by department
compute sum(sales), avg(sales), min(sales), max(sales)
  by department
compute sum(sales), avg(sales)
```

Results columns:

dbnumcols(dproc) = 4 -- This query will return 4 columns: department, year, month and sales

dbcolname(dbproc, 1) = "department" -- The name of the first column

dbcoltype(dbproc, 1) = SYBCHAR -- The datatype of the first column

dbcollen(dbproc, 1) = 10 -- The **maximum** length of the first column

dbdatlen(dbproc, 1) -- The answer will depend on the actual length for department for each row. If "toys" is returned, then dbdatlen = 4.

Compute columns:

dbnumcompute(dbproc) = 2 -- Two compute clauses in this query

dbnumalts(dbproc, 1) = 4 -- In the first compute there are 4 aggregate operations: sum, avg, min, and max

dbaltname(dbproc, 1, 3) = "" -- The third aggregate in the first compute has no title. In this example, none of them do.

dbaltop(dbproc, 1, 3) = MIN -- The third aggregate in the first compute is a min aggregate

dbalttype(dbproc, 1, 3) = SYBMONEY -- The datatype of the third aggregate of the first compute returns

dbaltlen(dbproc, 1, 3) = 8 -- The maximum length of the third aggregate in the first compute is 8 bytes

dbaltcolid(dproc, 2, 1) = 4 -- The first aggregate in the second compute refers to the 4th column in the query results -- sales.

dbbylist(dbproc, 1, &size) = {1} -- size is set to 1 and a pointer to an array of BYTEs is returned. size is the size of the bylist -- the number of elements in the array. The array has size 1 since the bylist of the first compute is department -- the first column in the results.

Using dbdata

- **Function**

  Allows access to data returned from the SQL Server without having to set up program variables and binds

  Returns a read-only BYTE pointer to the data

  Does not provide NULL terminators or conversion

- **Syntax**

C:	dbdata (dbproc, column#)
Fortran:	return = fdbdata(dbproc, column#, variable)

- **Fortran differences**

  Data is copied to the variable; pointers are not used

  Database and variable data types must match

  Returns FAIL if no column, or if NULL data

- **Example of C usage**

  Assume you are processing the results of a command such as: select id from sysobjects

```
printf( "object id is %d",*((int *) dbdata(dbproc,1)));
```

dbdata – Handling Nulls

- **How to recognize NULL data**

  Fortran returns FAIL if there is NULL or no column

  C returns a NULL pointer if there is NULL or no column

  The only way to know if the data is NULL is to test the data length for 0

- **Syntax**

  ```
  data_length = dbdatlen(dbproc, col#)
  data_length = fdbdatlen(dbproc, col#)  (fortran)
  ```

- **Why do we need to know if data is NULL?**

  To use defaults rather than NULLs, such as "not entered"

  To use printf, which will not accept the NULL pointer for NULL data

dbdata – Handling Nulls (cont'd)

- **Example of usage**

  Assume "select id from sysobjects":

  C:

```
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
     if (dbdatlen(dbproc,1) != 0)
          printf ("object id: %d\n",
                         *( (int *) dbdata(dbproc,1)) );
}
```

  Fortran:

```
do while( fdbnextrow(dbproc) .ne. NO_MORE_ROWS )
     if ( fdbdatlen(dbproc, 1) .ne. 0) then
          call fdbdata(dbproc, 1, objid)
          type *, 'object id: ', objid
     end if
end do
```

- **Notes**

  Testing for NULLs is not necessary using dbbind because dbbind converts NULLs

  C: Don't forget to "cast" the pointer to the right type

dbdata – Handling Strings

- **Finding the length of the string**

  Use dbdatlen to find the length of the string in order to determine where to put the NULL byte if needed

- **Example**

```
char  objname [40];
strncpy (objname, dbdata(dbproc,1), dbdatlen(dbproc,1));
objname [dbdatlen(dbproc,1)]='\0';
printf ( " %s ", objname);
```

- **Notes**

  You CANNOT add the NULL byte by calculating an offset to dbdata based on dbdatlen and then putting a NULL byte in the DBPROCESS structure.

  MONEY or DATETIME types must be converted to strings in order to use printf; they may be more easily managed using dbbind

  In general, use dbdata only when necessary.

Converting Data

- **When is this needed?**

  When you are not using dbbind, and want to make MONEY or DATETIME printable

  When you are using dbbind, but want to manipulate the data in some way and then return it to another database data type

  Whenever you want to convert one DB–Library datatype to another

dbconvert

- **Syntax**

```
dbconvert(dbproc, source_type, source_ptr,
          source_length, destination_type,
          destination_ptr, dest_length)
```

- **Parameters**

Source or Destination Types: SYBxxxx
(not the data type used in the definition, such as DBINT). See documentation under **types**.

Source Length: 0 means convert a null
                >0 length of variable strings
                −1 null terminated string or fixed type

Destination Length; −1 means use as much space as the data requires and terminate with nulls

- **Example (C)**

```
dbconvert(dbproc,SYBMONEY, dbdata(dbproc,1), -1,
          SYBCHAR, buffer, -1)
```

- **Example (Fortran)**

```
CHARACTER*8 var1
INTEGER*4 var2
...
call fdbdata(dbproc, 1, var1)
call fdbconvert(dbproc, SYBCHAR, var1, 8, SYBINT, var2,
2                    NULL)
```

Table of Matching Types

parameter in dbbind	parameter in dbconvert and bcpbind	C program datatype	Fortran program datatype
TINYBIND	SYBINT1	DBTINYINT	LOGICAL*1
SMALLBIND	SYBINT2	DBSMALLINT	INTEGER*2
INTBIND	SYBINT4	DBINT	INTEGER*4
CHARBIND	SYBCHAR	DBCHAR	CHARACTER*(*)
STRINGBIND	SYBCHAR	DBCHAR	CHARACTER*(*)
NTBSTRINGBIND	SYBCHAR	DBCHAR	CHARACTER*(*)
VARYCHARBIND	SYBCHAR	DBVARYCHAR	RECORD /VARYCHAR/
BINARYBIND	SYBBINARY	DBBINARY	CHARACTER*(*)
BITBIND	SYBBIT	DBBIT	LOGICAL*1
DATETIMEBIND	SBYDATETIME	DBDATETIME	CHARACTER*8
MONEYBIND	SYBMONEY	DBMONEY	CHARACTER*8
FLT8BIND	SYBFLT8	DBFLT8	REAL*8
VARYBINBIND	SYBBINARY	DBVARYBIN	RECORD /VARYBIN/

Comparing **dbbind** & **dbdata** - C

- Processing data using **dbbind**

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
extern int err_handler( );
extern int msg_handler( );
main( )
{
        DBPROCESS      *dbproc;
        LOGINREC       *login;
        STATUS    results;
        DBINT     royalty;
        DBCHAR    title[81], price[9];
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin( );
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, "select title, price, royalty from dbo.titles");
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("dbsqlexec failed\n");
                dbexit( );
                exit(ERREXIT);
        }
        dbresults(dbproc);
        dbbind (dbproc, 1, STRINGBIND, 0, title);
        dbbind (dbproc, 2, STRINGBIND, 0, price);
        dbbind (dbproc, 3, INTBIND, 0, &royalty);
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
        {
                printf (" %s\n $%s  %d\n", title, price, royalty);
        }
        dbexit( );
        exit(STDEXIT);
}
```

Comparing dbbind & dbdata – C

- **Processing data using dbdata and dbconvert**

```
/* include statements */
extern int err_handler( );
extern int msg_handler( );
main( )
{
        DBPROCESS       *dbproc;
        LOGINREC        *login;
        DBCHAR          title[81], price[10];
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin( );
        dbproc = dbopen (login, NULL);
        dbcmd(dbproc, "select title,price,royalty from dbo.titles");
        if (dbsqlexec(dbproc) == FAIL)
        {
                printf("error in dbsqlexec\n");
                dbexit( );
                exit(ERREXIT);
        }
        dbresults(dbproc);
        while (dbnextrow(dbproc) != NO_MORE_ROWS)
        {
                dbconvert(dbproc,SYBCHAR, dbdata(dbproc,1),
                                dbdatlen(dbproc,1), SYBCHAR, title, -1);
                dbconvert(dbproc,SYBMONEY, dbdata(dbproc,2),
                                -1, SYBCHAR, price,-1);
                if ( dbdatlen(dbproc,3) != 0)
                        printf ("%s\n $%s  %d\n", title, price,
                                        *((int *) dbdata(dbproc,3)) );
        }
        dbexit( );
        exit(STDEXIT);
}
```

Comparing **dbbind** and **dbdata** – Fortran

- **Processing data using fdbbind**

```
Program Bind
include ('fsybdb')
INTEGER*4              dbproc, login, royalty, results
CHARACTER*81           title
CHARACTER*10           price
EXTERNAL               err_handler, msg_handler
call fdberrhandle(err_handler)
call fdbmsghandle(msg_handler)
login = fdblogin( )
dbproc = fdbopen(login, NULL)
call fdbcmd(dbproc, 'select title, price, royalty from dbo.titles')
if (fdbsqlexec(dbproc) .eq. FAIL) then
      type *, 'fdbsqlexec failed'
      call fdbexit( )
      call exit
end if
call fdbresults(dbproc)
call fdbbind(dbproc, 1, STRINGBIND, 0, title)
call fdbbind(dbproc, 2, STRINGBIND, 0, price)
call fdbbind(dbproc, 3, INTBIND, 0, royalty)
results = fdbnextrow(dbproc)
do while (results .ne. NO_MORE_ROWS)
      type *, title
      type *, '$', price, '  ', royalty
      results = fdbnextrow(dbproc)
end do
call fdbexit( )
call exit
END
```

Comparing dbbind and dbdata - Fortran

- **Processing data using fdbdata and fdbconvert**

```
program test
include '(fsybdb)'
INTEGER*4       dbproc
INTEGER*4       login
INTEGER*4       retcode
CHARACTER*80 in_title
CHARACTER*8  in_price
CHARACTER*8  in_royalty
CHARACTER*80 title
REAL*8          price
INTEGER*4       royalty

EXTERNAL    msg_handler, err_handler
call fdbmsghandle(msg_handler)
call fdberrhandle(err_handler)
login = fdblogin()
dbproc = fdbopen(login,NULL)
if (dbproc .eq. NULL ) then
      call exit
end if
call fdbcmd(dbproc, ' select title, price, royalty from dbo.titles')
retcode = fdbsqlexec(dbproc)
if (retcode .eq. FAIL) then
      print *, 'Error in fdbsqlexec'
      call fdbexit ( )
      call exit
end if
retcode = fdbresults(dbproc)
do while (fdbnextrow(dbproc) .ne. NO_MORE_ROWS)
      call fdbdata(dbproc, 1, in_title)
      call fdbconvert(dbproc, SYBCHAR, in_title, -1, SYBCHAR,
2           title, -1)
```

```
          call fdbdata(dbproc, 2, in_price)
          call fdbconvert(dbproc, SYBMONEY, in_price, -1,
SYBFLT8,
     2         price, -1)
          if (fdbdatlen(dbproc, 3) .ne. 0) then
               call fdbdata(dbproc, 3, in_royalty)
               call fdbconvert(dbproc, SYBINT, in_royalty, -1,
SYBINT,
     2         royalty, -1)
               print *, title
               print *, '$', price, ' ', royalty
          end if
     end do
     call fdbexit()
     call exit
     END
```

Debugging Techniques

- **Failures from dbsqlexec**

  Try the command using ISQL

  Dump the command buffer using:

dbstrlen(dbproc)	no. of bytes in buffer
dbstrcpy(dbproc...)	copies bytes into variable
dbgetoff(dbproc...)	checks for SQL key words
dbgetchar(dbproc,n)	returns one byte of information from the buffer

- **Error Handling**

  Be familiar with the various error codes from DB–Library and the SQL Server. You can handle each of them any way you want to in your error and message handler code.

Handling Deadlock

- **Message number returned for deadlock**

  1205

- **In your SQL Server Message Handler**

  Check for this message number

  If deadlock, set a global flag, or use dbsetuserdata to set a flag in the dbproc

- **In your routine which did the dbsqlexec**

  Should check for FAIL on dbsqlexec

  If FAIL, check a flag which can be set in the error handler using dbsetuserdata, and if deadlock flag was set, rerun the dbsqlexec

  See documentation for parameters to dbsetuserdata, dbgetuserdata

- **Example (C)**

  See DB–Library reference manual, under dbsetuserdata.

Other Techniques

- **Use Stored Procedures**

  More work can be done at the Server end

  Stored procedures are faster than sending the same SQL code each time since recompilation is not needed

  Stored procedures can be modified without requiring a recompilation or re–link of your program (unless you change the results which are returned!)

- **Use dbbind instead of dbdata whenever possible**

  Makes programming simpler, more straightforward

  Generally speaking dbbind has the same if not better space/time efficiency as dbdata

- **Make full use of SQL features** (Transact SQL)

  Conditional code in SQL (if)

  Data conversion (select convert(varchar(20), date) ...)

  Retrieving substrings (select substring(...))

  Makes debugging, coding more effective

- **Have Fun!**

SYBASE

Module 5

Bulk Copy Functions

Objectives

- Review/understand the function of bulk copy
- Identify the two main ways of using bulk copy from a program
- Learn the commands to transfer data from program variables to the data base
- Learn the commands to transfer data from a file to the data base

What is Bulk Copy?

- **Function**

  Allows users to copy data into or out of database tables

  Provides facility for high-speed loading of data from a host file or program variables into the SQL Server

  Allows you to move only portions of a database or file

- **Comparison to equivalent SQL functions**

  Bulk copy out and SQL select are similar speeds but bulk copy can go directly to a pre-defined file format

  Bulk copy in and SQL insert differ if you use high speed bulk copy in which turns off logging

  Rules and triggers are not enforced on a bulk copy in; defaults are always enforced

- **Three equivalent interfaces**

bcp: stand-alone program to copy in/out a file

DWB/copy table: same as bcp with a visual interface

BCP-Library: a subroutine level interface callable from an application program

Stand-alone and DWB bulk copy both use the BCP-Library functions

High Speed Copy In

- **Description**

  Invoked when copying data into a non–indexed table

  Data is not logged and goes directly to database

  Page allocation is logged for recovery purposes

  Database option "select into/bulkcopy" must be turned on

  Database checkpointed on completion

  Copying data in always appends to the database

- **Recovery issues**

  If system crashes during bulk copy, pages are de–allocated: no new data remains in the database

  If system crashes after bulk copy all the data is in the database

  Be sure to dump the database after a copy in

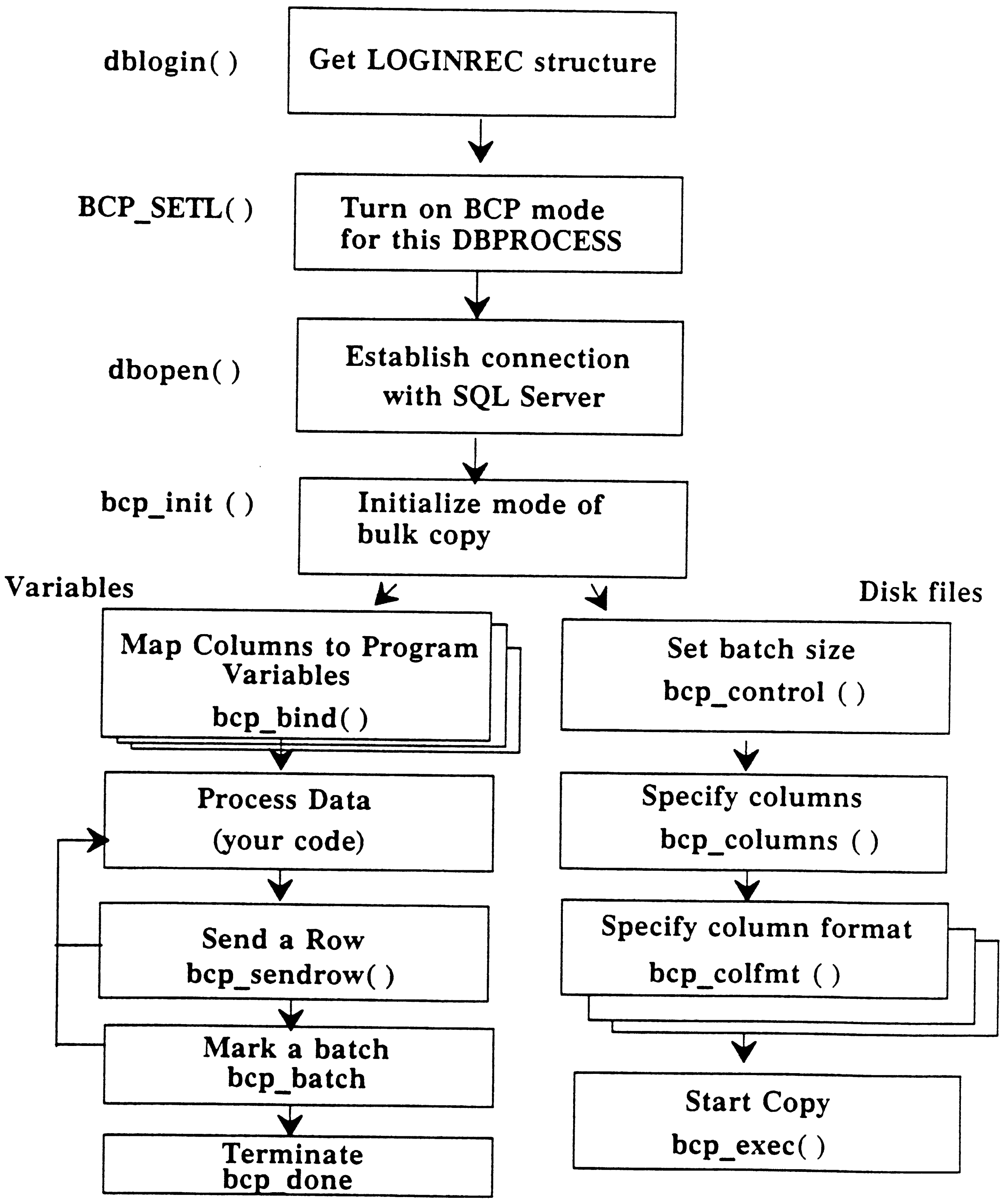
- **Copying Batches of data**

  Setting a batch size forces each batch to be a transaction (begin/commit) with a checkpoint at the end of each batch

  Recovery is guaranteed up to the last completed batch prior to system crash

  Typical batch size: 10000 rows

Programming Overview



Getting Started

- **Library functions**

```
BCP_SETL(login, TRUE)
```

set up the login structure for BCP

```
retcode = bcp_init(dbproc, table, hostfile,errorfile,direction)
```

initialize the bulk copy

- **Parameters for bcp_init**

table	table name in the database
hostfile	name of file NULL means you are using variables
errorfile	file name for logging errors **or** NULL means no error file
direction	DB_IN DB_OUT

- **Example of usage**

```
login = dblogin ()
BCP_SETL(login, TRUE)
dbproc = dbopen (login, NULL)
retcode = bcp_init ( dbproc, "pubs.authors",
                     "authors.file", NULL, DB_OUT)
if (retcode == FAIL)....etc
```

Bind Columns to Variables

- **Syntax**

  ```
  ret = bcp_bind (dbproc, var_addr, prefix_length,var_length
                          terminator, term_length, type, col_#)
  ```

- **Function**

  Bind a variable to a column in the table specified by bcp_init

  Must be issued once for each column you are copying into

  Converts data if variable type doesn't match column type in the database

bcp_bind, cont'd

- **Syntax**

```
ret = bcp_bind (dbproc, var_addr, prefix_length,
                var_length, terminator,
                term_length, type, col_#)
```

- **Interesting Parameters**

prefix_length	used in non–C string specifiers 0 means no prefix is in the data
var_length	does not include the prefix or terminator –1 means length determined by prefix or terminator
type	0 means don't do conversion SYBxxx means type of program variable
col_#	column number in the table

- **Specifying a terminator**

Must be a BYTE pointer to a sequence of bytes

To specify a NULL terminator (C)

    terminator "", term_length 1

To specify NO terminator (Fortran)

    terminator NULL, term_length 0

Examples - bcp_bind

- **String variable bound to string column**

```
char name[10];
char terminator = " ";        /* the null string */
bcp_bind(dbproc, name, 0, -1, (BYTE *) terminator,
            1, 0, 2);
```

var_addr	name
prefix_length	0 (none)
var_length	-1 (copy till terminator reached)
terminator	"" (null-terminator)
term_length	1 (= 1 byte = 1 character)
type	0 (no conversion)
col_#	2 (second column in the table)

- **Int variable bound to int column**

```
DBINT id;
bcp_bind(dbproc, &id, 0, -1, (BYTE *) NULL, 0, 0, 1);
```

var_addr	&id
prefix_length	0 (none)
var_length	-1 (use default length of data)
terminator	NULL
term_length	0 (no terminator)
type	0 (no conversion)
col_#	1 (first column in the table)

- **Int variable bound to string column**

```
DBINT id;
bcp_bind(dbproc, &id, 0, -1, (BYTE *) NULL, 0,
               SYBINT, 1);
```

Sending the data

- **Related functions**

bcp_sendrow(dbproc)	Send data to Server
bcp_batch(dbproc)	Force a checkpoint (end transaction) of the data
bcp_done(dbproc)	Commit the transaction, and checkpoint the database

- **Notes**

bcp_batch is completely optional, but recommended if you are copying a lot of data into a table without indexes

Batch size is completely dependent on the program code and how often it calls bcp_batch

bcp_done is required to terminate the process.

- **Example**

See the documentation for bcp_bind, or fbcp_bind

Bulk copy using host files

- **Function**

  Move data (at high speed) from or to a host file

- **Features**

  File contents can be completely described using BCP-Library routines

  You can specify which columns in the file, and which rows or columns in the table, are involved in the copy

- **Library functions**

  bcp_init (...specify the file name)

  bcp_control (dbproc, BCPBATCH, n) (set batch size)

  bcp_columns(dbproc, no._of_cols)
  specify the number of columns in the file

  bcp_colfmt(dbproc, col_#, type, prefixlen, col_length, terminator, term_length, table_col#)

  Note its resemblance to bcp_bind, but in this case we are describing the characteristics of the column in the file rather than a program variable

  bcp_exec(dbproc, &rows_copied)
  do it!

Example – copy out to file

- **Scenario**

  Copy out the first name, last name and phone number from the authors table into a file, in order to make a phone list

- **Code excerpt**

```
bcp_init(dbproc, "authors", "authors.out", "bcp.error", DB_OUT);
bcp_columns(dbproc, 3);
bcp_colfmt(dbproc, 1, SYBCHAR, 0, 20, "\t", 1, 3);
bcp_colfmt(dbproc, 2, SYBCHAR, 0, 40, "\t", 1, 2);
bcp_colfmt(dbproc, 3, SYBCHAR, 0, 12, "\n", 1, 4);
if (bcp_exec(dbproc, &rows)==FAIL)
{
     printf("not good!\n");
     dbexit( );
}
```

- **Fortran**

Summary

- **Bulk copy provides methods for moving data in or out of the data base at high speeds**
- **There are two methods for copying data in – from disk files or from program variables**
- **List of functions for working with program variables**

  bcp_init

  bcp_bind

  bcp_sendrow

  bcp_batch

  bcp_done

- **List of functions for working with disk files**

  bcp_init

  bcp_control

  bcp_columns

  bcp_colfmt

  bcp_exec

Lab Exercise: Bulk Copy

1. Create a table using SQL, with 2 columns, both of which are integers. The first column will be a counter, the second column will be a random number. (ie.: **create table test (counter int, random int)**).

2. Write a program which will use bulk copy to fill the table. For each row, insert a counter into the counter column which starts at 1 and increments by 1. Generate a random number for the second column (for C, use the procedure **random**(); in Fortran, use the function **irand**().) Send over 2000 rows of data in batches of 100. Test what you have done by using ISQL and doing a **select count**(*) from the table.

Optional:

3. Write a program which copies the data from your little table out to a disk file. The file should be ascii data, with the first column follwed by a tab, and the second column followed by a newline.

Lab Answer

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
int err_handler();
int msg_handler();
main()
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        RETCODE  results;
        DBINT          counter;
        DBINT   rndom;
        int            batch;
        batch = 0;
        counter = 0;
        rndom = 0;
        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin();
        BCP_SETL(login, TRUE);
        dbproc = dbopen (login, NULL);
        results = bcp_init(dbproc, "mytable",NULL, NULL, DB_IN);
    results= bcp_bind(dbproc, &counter, 0,-1,(BYTE *)NULL, 0, 0, 1 );
    results = bcp_bind(dbproc, &rndom, 0, -1, (BYTE *)NULL, 0, 0, 2);

        while (counter++ < 2000)
        {
                        if(batch ++ > 100)
                                {
                                bcp_batch(dbproc);
                                batch = 0;
                                printf( "batch complete \n");
                                }
        rndom = rand();
        bcp_sendrow(dbproc);
        }
        bcp_done(dbproc);
        dbexit();
        exit();
}
```

Optional Lab Answer

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

int err_handler();
int msg_handler();
main()
{
        DBPROCESS *dbproc;
        LOGINREC *login;
        RETCODE  results;
        DBINT       counter;
        DBINT   rndom;
        int            rows;
        counter = 0;
        rndom = 0;

        dberrhandle(err_handler);
        dbmsghandle(msg_handler);
        login = dblogin();
        BCP_SETL(login, TRUE);
        dbproc = dbopen (login, NULL);

        results = bcp_init(dbproc, "mytable","my.out", NULL, DB_OUT);
        bcp_columns(dbproc,2);
        bcp_colfmt(dbproc,1,SYBCHAR,0,20, "\t",1,1);
        bcp_colfmt(dbproc,2,SYBCHAR,0,20, "\n",1,2);
        bcp_exec(dbproc,&rows);

        dbexit();
        exit();
}
/*  */
```

SYBASE

Module 6

Two-Phase Commit

Objectives

- Understand what the two–phase commit service is used for and how it works from a user perspective

- Describe the general structure of code which uses two–phase commit service

- Learn the DB–Library functions which support two–phase commit protocol

What is Two-Phase Commit?

- **Definitions**

  Two-phase commit is a mechanism to treat separate transactions (which may be on separate SQL Servers) as if they were one transaction

  The goal is to guarantee correct recovery for all transactions participating in the two-phase commit "event"

- **Why is it necessary?**

  Recovery is normally based on the log entries for a specific database on a specific SQL Server

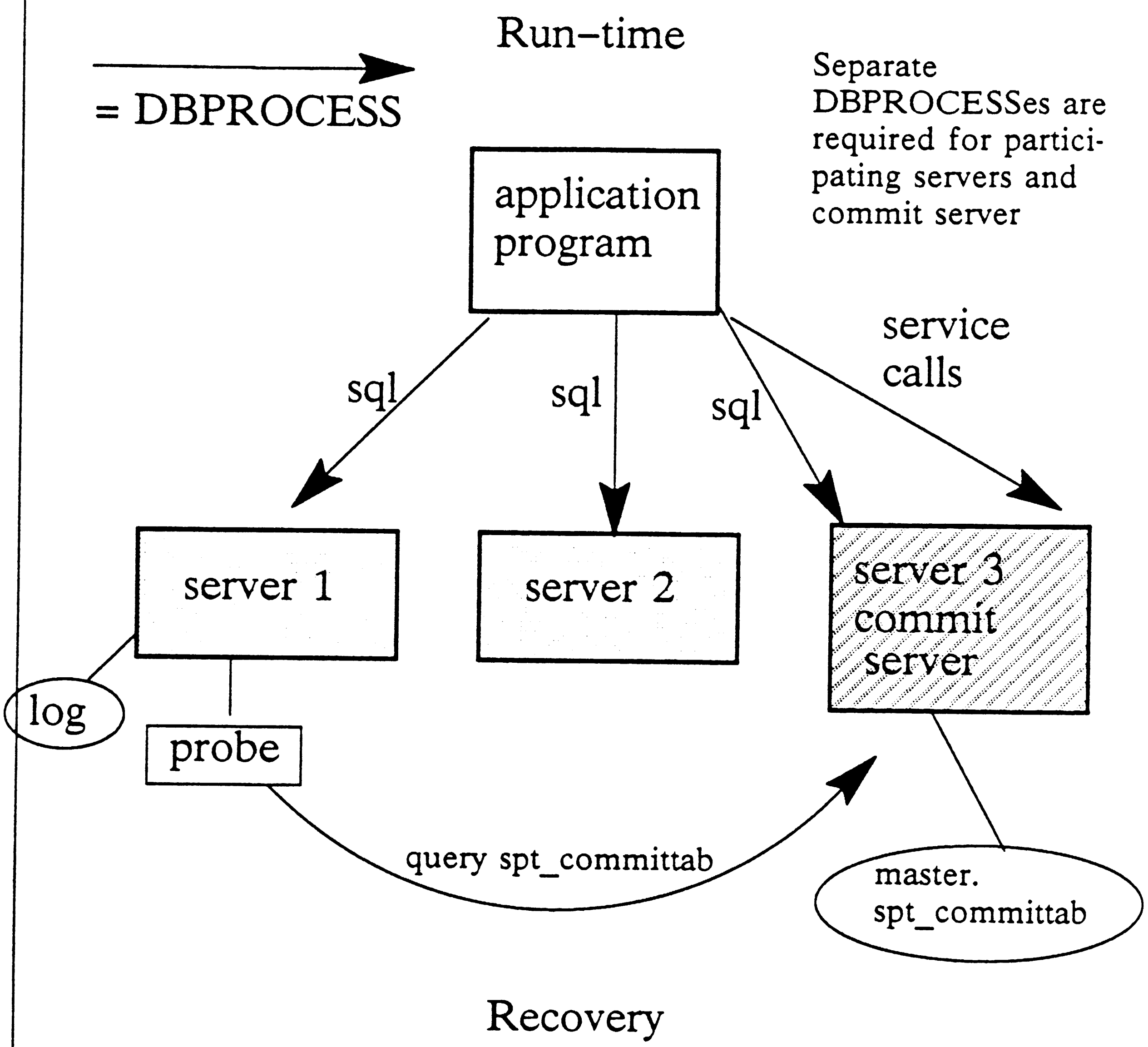
  Two-phase commit transactions depend on additional information kept by the commit service SQL Server

- **How is it accomplished?**

  Transactions are committed in two phases:
  Prepare phase – guarantees they can commit
  Commit phase – actual commit takes place

  Recovery (rollback or commit decision) for any transaction which reached the Prepare phase will be based on status in the Commit Service rather than the local log

Overall Flow



If server 1 fails prior to commit, **probe** is started if the log indicates transaction was in the Prepare stage at the time of failure

Components of Two–Phase Commit

- **Runtime Component**

  A programming protocol implemented using DB–Library and the SQL “prepare” statement

  Protocol insures that prior to the actual commit of the set of transactions, each transaction in the set can commit

- **Bookkeeping Function**

  One server is designated as the commit service

  It provides a central location for all processes to find information about the transaction

  It maintains the status of the two–phase commit transaction on–line in master.spt_committab

- **Recovery Time Components**

  SQL Server Recovery code guarantees that if the transaction has reached the Prepare stage, the commit (or rollback) will take place across all servers participating in the event

  Any server which needs to recover a two–phase commit transaction will start the Probe program which then queries the commit server for the transaction status

Probe

- **What is it?**

  A DB-Library program located in $SYBASE/bin on every system with a server

  Each server also has a SQL Server account called "probe" which allows the probe program to log in

  Thus any server could function as a commit server

- **How is it used?**

  Server running recovery finds a log entry for a two-phase commit transaction in Prepare State

  Server decodes the transaction id and the commit server name from the log and passes these to Probe

  Probe connects to the commit server (using the query port for the server in the interfaces file) and sends a query to read the status from spt_committab

  Probe returns the status (commit or rollback) to the original server and then logs out

- **What is needed for it to work?**

  Entry in the interfaces file on the recovering server's machine for the commit server

  Each server as well as the commit service requires its own dbproc

Programming Components

- SQL Code (sent via **dbsqlexec**) for each server

  begin transaction name

  name is generated by calls to DB–Library

  update, delete, etc,

  these are the normal statements used in the transaction

  prepare transaction

  issued prior to any commits to verify that all participants can commit

  affects recovery: failure prior to prepare always means local rollback; failure after prepare always means check with the commit service

  If not everyone can successfully prepare, the application program would abort the transaction and rollback

  commit transaction

  must be done AFTER commit service has been informed the commit will take place

Commit Service Calls

- **start_xact**

  Informs the commit service that a distributed transaction is about to start

- **commit_xact**

  Ends first phase and begins phase two of the distributed transaction

  If the commit_xact is issued, then the transaction is guaranteed to be committed on all participating servers

  Function is to change the status in spt_committab for this transaction so that if a server queries the commit server, the status will show "commit"

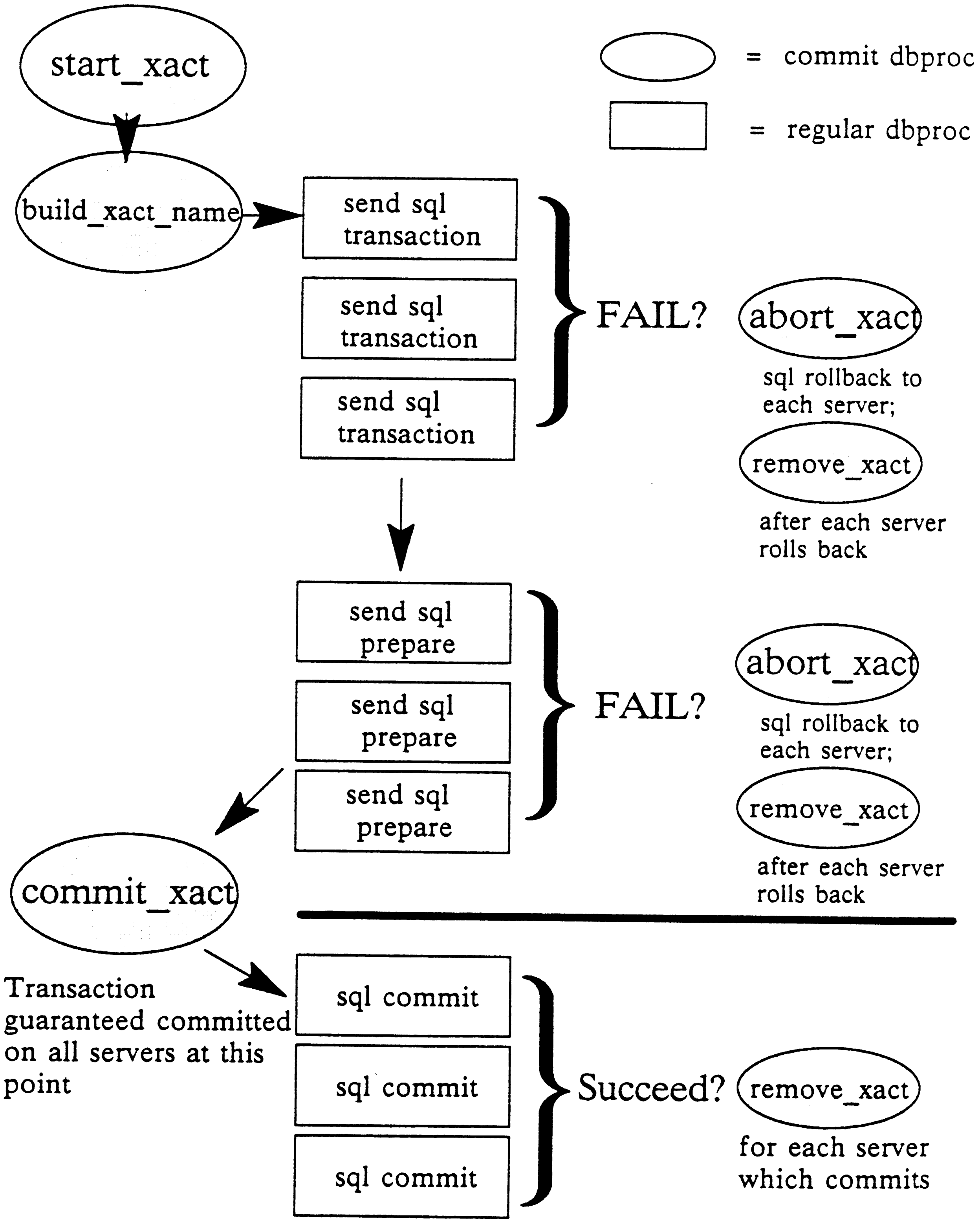
- **abort_xact**

  If any of the participants cannot carry out the transactions or fail to prepare, the abort_xact changes the status in spt_committab to show "roll-back"

- **remove_xact**

  As each process completes (commit or rollback) it issues this to indicate it is finished with this transaction; when all participants complete, the information for this transaction is removed from spt_committab

Overall Program Flow



Building the Program

- **Getting Started**

  Select a commit service server

  Open and/or verify connections to all participating servers
  (dbopen, open_commit)

  Get a transaction name and id to identify transaction to servers and commit service
  (start_xact, build_xact_string)

- **Sending the transaction**

  Build strings containing the sql, then send it
  (dbcmd, dbfcmd, dbsqlexec)

  Test for errors from each server as needed

- **Committing**

  Send and test for successful prepare
  (dbcmd, dbsqlexec)

  Commit to service or abort
  (commit_xact, abort_xact)

  Commit on each server and remove from active state
  (dbcmd, dbsqlexec)
  (remove_xact)

- **Close the connections**

  (close_commit, dbexit)

Getting Started

- dbproc = open_commit(login, "server_name")

  Establishes a DBPROCESS for use in communicating with the commit server

  Uses the standard DB_Library login structure; returns NULL if the connection fails

- **Commit Service Server**

  Any server can be a commit service server

  Server name is indicated in the second parameter to open_commit

  Interfaces file for all the participating servers must contain at least the query entry for the commit server

- **Sample Interfaces file for "Practice" if "Sybase" is the commit server:**

```
#
SYBASE
        query tcp sun-ether godzilla 2001

#
PRACTICE
        query tcp sun-ether godzilla 5000
        master tcp sun-ether godzilla 5000
        console tcp sun-ether godzilla 5001
        debug tcp sun-ether godzilla 5003
        trace tcp sun-ether godzilla 5004
```

Getting Started - sample code

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

int err_handler( );
int msg_handler( );
main( )
{
      DBPROCESS *dbproc_server1;
      DBPROCESS *dbproc_server2;
      DBPROCESS *dbproc_commit;
      LOGINREC *login;
      int           commid;
      char          cmdbuf[256];
      char          xact_string[128];

      RETCODE ret;
      RETCODE ret2;
      dberrhandle(err_handler);
      dbmsghandle(msg_handler);

      login = dblogin( );
      dbproc_server1 =  dbopen (login, "SYBASE");
      dbproc_server2 = dbopen (login, "PRACTICE");
      dbproc_commit = open_commit (login,"SYBASE");
      if (dbproc_server1==NULL || dbproc_server2 == NULL ||
                  dbproc_commit ==NULL)
      {
                  printf (" connections failed!\n");
                  exit (-1);
      }
etc...
}
/*  */
```

Starting the Transaction

- **id = start_xact(dbproc_commit, "application name", "transaction name", number_of_servers)**

  The id returned is used to identify this transaction in all subsequent calls to the commit service

  In addition, the id and the transaction name are used to build the transaction name used in SQL

  The number of servers is used to determine when the transaction is completed by all servers (it is decremented by **remove_xact**)

  At this point the distributed transaction is considered started

- **build_xact_string ("transaction_name", "commit_server_name", id, xact_string)**

  xact_string represents the address of space which you have allocated for the transaction name

- **dbfcmd(dbproc,"begin transaction %s", xact_string)**
  dbsqlexec(dbproc)

  Start the transaction using the newly built name. This must be done for each participating server

  More efficient code would build the **dbfcmd** string in a buffer then use **dbcmd(dbproc,buffer)** to send the same command to each server

Starting the transaction (code excerpt)

```
......
        DBPROCESS *dbproc_server1;
        DBPROCESS *dbproc_server2;
        DBPROCESS *dbproc_commit;
        LOGINREC *login;
        int          commid;
        char         cmdbuf[256];
        char         xact_string[128];

        login = dblogin( );
        dbproc_server1 = dbopen (login, "SYBASE");
        dbproc_server2 = dbopen (login, "PRACTICE");
        dbproc_commit = open_commit (login,"SYBASE");

/* ... code to test for failure of opens goes here  */

        commid = start_xact(dbproc_commit, "labdemo", "test",2);
        build_xact_string ("test", "SYBASE", commid, xact_string);

/*      build  command buffer */

        sprintf(cmdbuf, "BEGIN TRANSACTION %s",xact_string);
        dbcmd(dbproc_server1,cmdbuf);
        dbsqlexec(dbproc_server1);
        dbcmd(dbproc_server2,cmdbuf);
        dbsqlexec(dbproc_server2);
.......
```

Commit and Close

```
if (commit_xact(dbproc_commit,commid) ==FAIL)
{
      abort_xact(dbproc_commit, commid);
      error_function( ); /* rollback and remove each participant */
}
sprintf(cmdbuf, "COMMIT TRANSACTION");
dbcmd(dbproc_server1,cmdbuf);
ret1= dbsqlexec(dbproc_server1);
if (ret1 != FAIL)
      remove_xact(dbproc_commit,commid,1);
dbcmd(dbproc_server2,cmdbuf);
ret1= dbsqlexec(dbproc_server2);
if (ret1 != FAIL)
      remove_xact(dbproc_commit,commid,1);
close_commit(dbproc_commit);
printf( " we made it!\n");
dbexit( );
}
```

Demonstration

- **What is it doing?**

  Performing the same update identical information in databases on two different servers

  Each database has the pubs database; the program is updating the price for one of the titles

- **More typical Uses of Two-Phase commit**

  Update different but related information on two different databases on different servers

  Note that updating two databases on one server at once in a transaction implicitly does a two-phase commit for you

Summary

- **Two–phase commit supports distributed transactions**

  You write the code, Sybase guarantees the rest

- **Special functions**

  open_commit
    start_xact
    build_xact_string
    SQL PREPARE
    commit_xact
    abort_xact
    remove_xact
  close_commit

Sample Program

```
/*  Lab Demo of Two Phase Commit  */

#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>

int err_handler( );
int msg_handler( );

char        cmdbuf[256];
char        xact_string[128];

main()
{

            DBPROCESS *dbproc_server1;
            DBPROCESS *dbproc_server2;
            DBPROCESS *dbproc_commit;
            LOGINREC *login;
            int         commid;

            RETCODE ret_server1;
            RETCODE ret_server2;

            dberrhandle(err_handler);
            dbmsghandle(msg_handler);

            printf("Lab demo of Two Phase Commit\n");

            login = dblogin( );
            DBSETLUSER(login, "sa");
            dbproc_server1 = dbopen (login, "SYBASE");
            dbproc_server2 = dbopen (login, "PRACTICE");
            dbproc_commit = open_commit (login, "SYBASE");

            if (dbproc_server1 == NULL ||
                dbproc_server2 == NULL ||
                dbproc_commit  == NULL)
            {
                        printf (" connections failed!\n");
                        exit (-1);
            }

            /* use the pubs database */
            sprintf(cmdbuf, "use pubs");
            dbcmd(dbproc_server1, cmdbuf);
            dbsqlexec(dbproc_server1);
            dbcmd(dbproc_server2, cmdbuf);
            dbsqlexec(dbproc_server2);
```

```
/* start the distributed transaction on the commit service */
commid = start_xact(dbproc_commit, "labdemo", "test", 2);

/* build the transaction name */
build_xact_string ("test", "SYBASE", commid, xact_string);

/* build first command buffer */
sprintf(cmdbuf, "BEGIN TRANSACTION %s", xact_string);

/* begin the transactions on the different servers */
dbcmd(dbproc_server1, cmdbuf);
dbsqlexec(dbproc_server1);
dbcmd(dbproc_server2, cmdbuf);
dbsqlexec(dbproc_server2);

/* do various updates */
sprintf(cmdbuf, " update titles set price = $1.50 where");
strcat(cmdbuf, " title_id = 'BU1032'");
dbcmd(dbproc_server1, cmdbuf);
ret_server1 = dbsqlexec(dbproc_server1);
dbcmd(dbproc_server2, cmdbuf);
ret_server2 =dbsqlexec(dbproc_server2);
if (ret_server1 == FAIL || ret_server2 == FAIL)
{
        /* some part of the transaction failed */
        printf(" Transaction aborted, sql exec failed\n");
        abortall(dbproc_server1, dbproc_server2, dbproc_commit, commid);
}

/* find out if all servers can commit the transaction */
sprintf(cmdbuf, "PREPARE TRANSACTION");
dbcmd(dbproc_server1, cmdbuf);
dbcmd(dbproc_server2, cmdbuf);
ret_server1 = dbsqlexec(dbproc_server1);
ret_server2 = dbsqlexec(dbproc_server2);
if (ret_server1 == FAIL || ret_server2 == FAIL)
{
        /* one or both of the servers failed to prepare. */
        printf(" Transaction aborted, prepare failed\n");
        abortall(dbproc_server1, dbproc_server2, dbproc_commit, commid);
}

/* Commit the transaction */
if (commit_xact(dbproc_commit, commid) == FAIL)
{
        /* The commit server failed to record the commit */
        printf( " Transaction aborted because commit xact failed\n");
        abortall(dbproc_server1, dbproc_server2, dbproc_commit, commid);
        exit(-1);
}

/* the transaction has successfully committed.  Inform the different
** servers.
```

```
            */
            sprintf(cmdbuf, "COMMIT TRANSACTION");
            dbcmd(dbproc_server1, cmdbuf);
            if (dbsqlexec(dbproc_server1) != FAIL)
                        remove_xact(dbproc_commit, commid, 1);
            dbcmd(dbproc_server2, cmdbuf);
            if (dbsqlexec(dbproc_server2) != FAIL)
                        remove_xact(dbproc_commit, commid, 1);

            /* Close connection to Commit Server */
            close_commit(dbproc_commit);

            printf( "We made it!\n");
            dbexit( );
exit( );
}

/* Function to abort the distributed transaction */

abortall( dbproc_server1, dbproc_server2, dbproc_commit, commid )
DBPROCESS *dbproc_server1;
DBPROCESS *dbproc_server2;
DBPROCESS *dbproc_commit;
int         commid;
{
            /* some part of the transaction failed */

            /* inform the commit server of the failure */
            abort_xact(dbproc_commit, commid);

            /* roll back the transactions on the different servers */
            sprintf(cmdbuf, "ROLLBACK TRANSACTION");
            dbcmd(dbproc_server1, cmdbuf);
            if (dbsqlexec(dbproc_server1) != FAIL)
                        remove_xact(dbproc_commit, commid, 1);
            dbcmd(dbproc_server2, cmdbuf);
            if (dbsqlexec(dbproc_server2) != FAIL)
                        remove_xact(dbproc_commit, commid, 1);

            dbexit( );
            exit(-1);
}
```

Lab Exercise: Two phase commit

1. Create a simple table on the practice server and sybase servers.

   Write a two-phase commit transaction which inserts a row into both tables on both servers.