JAM/DBi

Please forward comments regarding this document to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

JAM/DBi

Contents

1 Introduction

1.1 What is JAM/DBi?

JAM/DB*i* provides an easy-to-use, standard, portable interface between JAM applications and a variety of popular SQL-based databases.

JAM is a development tool which provides a prototyping, development and testing environment for the rapid development of software applications.

SQL (standard query language) is a tool which provides end users with a non-procedural, easy-to-use means of accessing databases. SQL assumes little or no programming skills. SQL is also an emerging standard. This means that users can move from one machine, operating system and/or database to another with little or no retraining in database access methods.

JAM/DB*i* ties together the cost-effectiveness of JAM application development with the power of SQL-based databases. And, if the developer so chooses, it can all be done without ever writing a line of third-generation, procedural programming code.

A key feature of JAM/DB*i* is that it uses the query language syntax (SQL) of the database you are using. Instead of learning a new syntax, users continue to use the syntax with which they are already familiar.

JAM/DB*i* is easy to use, because data retrieval and update are accomplished through JAM data dictionary definitions. The user is not required to understand the lower level operation of the database or JAM. When data are retrieved (using an SQL SELECT) they are placed in data dictionary or screen elements whose names correspond to the database table column names. If column names do not match data dictionary elements, they may be "coerced" or mapped using database-supported column mapping (or alias mapping if a database does not support column mapping.) For updates (SQL INSERT or UPDATE), the user simply specifies the names of the data dictionary or screen elements to be inserted as host variables.

There are many advantages to the JAM/DB*i* solution:

- it makes linking application screens to databases trivial
- it eliminates the need to know the low level access routines of a database system
- it virtually guarantees portability of an application across many different hardware and operating system platforms
- it provides a standard means of moving applications from one database to another with no changes to screens and very few (if any) changes to SQL scripts
- it enables a developer to prototype an application with real links to a database without ever writing a line of code. Later the developer can go back to the same application and build in procedural calls to provide additional functionality.

1.2 What Makes Up JAM/DBi?

The media, file names and contents of your JAM/DB*i* files are dependent on the hardware and operating system for which you ordered JAM/DB*i*.

Every version of JAM/DB*i* should have the following files:

- one or more files (usually in binary format) that contain the object code of JAM/DB*i*;
- utilities;
- a makefile for a JAM/DB*i* version of `jxform`;
- a makefile for a JAM/DB*i* version of JAM;
- makefiles for utilities (if necessary).

The actual file names for your machine and operating system are described in Appendix A of this document.

1.3 What Is in This Document?

This document describes

- what JAM/DB*i* is;
- how to use JAM/DB*i* in building applications that use supported databases,

- and how to use JAM/DB*i* in specific hardware and operating system environments.

This document assumes that the reader has a basic knowledge of the target computer system, databases, JAM and `jxform`. Except for details related to building a JAM/DB*i* link, this manual does not provide any details about how to use `jxform` or JAM.

1.4 What Is the JAM/DBi Development Cycle?

Before you can do any development, you have to build a copy of JAM that includes the links to the database interface. This executable is `jamdbi`. Appendix A describes the procedure you should follow. You may also build a version of `jxform` with links to a database. This executable is `jxdbi`. Some versions of `jxdbi` running under PC-DOS will not have enough memory to do development. In such cases, you will have to use `jxform` without DBi linked in and test using `jamdbi`.

There are four steps to building a working application with JAM/DB*i*.

- First, you build a JAM screen using `jxform` or `jxdbi`. This involves defining JAM fields and providing an access path between a JAM application and JAM/DB*i*. In building a JAM screen, you have several ways to gain access to JAM/DB*i*. The quickest way is to make a screen entry call to JPL. The second way is to associate a JAM/DB*i* function with a keyboard key. The third means of access is through attached functions. The details of making these calls are presented below in a demonstration scenario.
- Second, you should add the fields that you defined in your JAM screens to the JAM data dictionary. This ensures that the data brought in from the database to one screen will be available in all other screens, too.
- Third, you need to create a set of JPL statements that are the actual SQL calls to the database. The name of the JPL file must be the same as the one specified in the JAM screen entry specification or in an associated key. We assume that you already have a set of database tables with some data to access.
- Fourth, you test your program.

1.4.1 A Development Scenario

In the next few pages, we will walk you through a complete JAM/DB*i* development cycle. For our scenario we assume you are using the ORACLE RDBMS with the following table:

MYDATA

with the following fields:

NAME ADDRESS CITY STATE.

To create MYDATA, go into SQLPLUS and enter:

```
create table mydata
(name char (30),\
address char (30),\
city char (15),\
state char(2));
```

You can insert several values into this table with the following statements:

```
insert into mydata values
('John Doe',\
'1312 Geary Blvd',\
'San Francisco','CA');
insert into mydata values
('Jane Roe',\
'505 West End Ave',\
'New York','NY');
insert into mydata values
('Edgar Woe',\
'3712 Rio Grande Blvd',\
'Albuquerque','NM');
insert into mydata values
('Amy Snow',\
'1400 Lakeshore Dr',\
'Chicago','IL');
```

Now exit SQLPLUS and go into `jxform` (or `jxdbi`).

Press <Shift PF5> and go into **FORMAKER**.

When prompted for the name of a form, enter MYTEST and press <XMIT>.

Press <PF3> to define a border and background colors. Depending on the characteristics of your terminal, assign any values you want but leave the size of the form as the default size.

Press <XMIT>.

Press <SHIFT PF1> to associate a function key with a set of SQL instructions. A window will appear with the names of the keyboard keys. Move your cursor down to PF1 and at the prompt for the name of an entry function enter:

```
^jpl myjpl.jpl
```

Press <XMIT>.

Move your cursor down two lines and over several columns.

Now draw in an underscore with 30 characters and, when finished, move back one space. Press <XMIT>.

Press <PF4> to define field attributes. Assign the following characteristics:

Type <A> and enter the field name NAME (upper case for ORACLE!).

Press <XMIT>.

Type <S> to change the size of the field to an array.

Tab to number of elements and enter "10" for an array of 10 occurrences.

Tab to horizontal and enter <N> (for "No").

Press <XMIT> to confirm these field attributes.

Type <E> (for "Exit") to return to the form window.

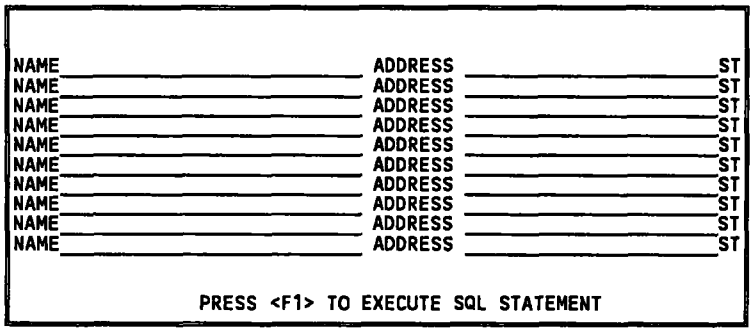
Move the cursor over a few spaces and repeat the same process for a field of 30 characters and name it ADDRESS.

It is important for ORACLE that the field names be entered in upper case. (Some databases do not care, and some want the entry to be in lower case. These considerations are described in Appendix B.)

Move your cursor to the next-to-the-last line on the screen and about 20 columns from the left and enter the following text:

```
PRESS <F1> TO EXECUTE SQL STATEMENT
```

You should have a screen that looks something like this when you press <SHIFT PF3>:



Press <PF1> to return to the form window.

Press <EXIT> and type <S> to save your form. When the "form saved" message appears, type <SPACE> and <E> to exit.

Using any text editor, create an ASCII text file called

```
'myjpl.jpl'
```

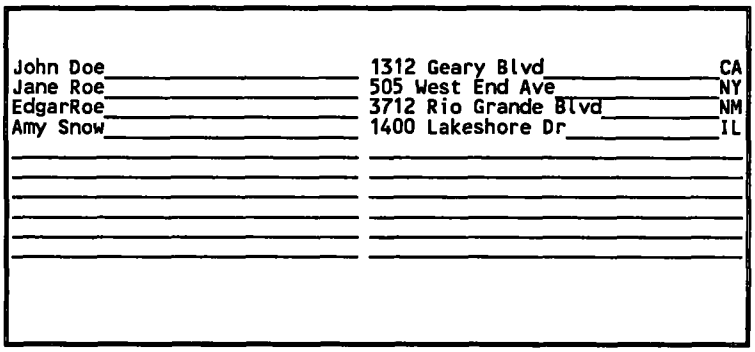
and enter the following lines:

```
DBMS LOGON username password
SQL SELECT NAME,ADDRESS,STATE FROM MYDATA
DBMS LOGOFF
RETURN 0
```

username and *password* are your name and password on your ORACLE RDBMS. (We assume you have been granted resources to create a table.)

Exit your editor and type JAMDBI MYTEST. When the screen comes up, press <PF1> to execute the JPL/SQL statements.

If all went well, your screen should look like this:



To exit, press <EXIT>.

2 Accessing JAM/DBi

There are two basic ways of accessing JAM/DB*i* functions: through JPL statements and direct C language function calls.

The advantage of using JPL is that

- it is easier to prototype an application;
- it is easier to see what is going on in a particular function;
- it is easier to change a function,
- and you do not have to compile anything in order to see your application run.

The advantage of using C calls is that

- the reading and first parsing of JPL statements are eliminated;
- your SQL calls are embedded and cannot be changed (or erased!) by end users,
- and your applications are more secure.

Both of these access methods are described below.

2.1 JPL Calls

There are two types of DB*i* statements in JPL. The JPL DBMS statements begin with the keyword DBMS and JPL SQL statements begin with the keyword SQL.

(Note: Under the ShareBase version of JAM/DB*i* there is an additional JPL verb - IDL - which works much like SQL. The IDL dependent features are described in Appendix B of the JAM/DB*i* ShareBase documentation.)

2.1.1 JPL DBMS Calls

DBMS statements are used for either database specific functions such as logging on, or for controlling the JAM/DB*i* environment such as setting error levels. Examples of JPL DBMS statements are:

```
DBMS LOGON DEMO JIM MYPASSWD
```

```
DBMS COUNT MYCOUNT
DBMS ERROR
```

2.1.2 JPL SQL Calls

SQL statements are used for standard SQL calls such as SELECT, INSERT, UPDATE, COUNT, etc. Examples of the use of JPL SQL statements are:

```
SQL SELECT NAME, ADDRESS, CITY FROM MYDATA
SQL SELECT NAME FOR UPDATE FROM EMP WHERE\
                    EMPLOY_NUM=678
```

2.2 Embedded C Calls

If desired, DBMS and SQL calls may be hidden and passed directly to JAM/DB*i*. However, the developers who do this must be careful to *test the return codes* and handle errors properly.

There are two functions that can be called, namely, **dbi_dbms** and **dbi_sql** (summarized below). The appropriate function call corresponds to whether the argument being passed is a DBMS or a SQL statement.

NAME

dbi_dbms -parse and execute a DBMS statement

SYNOPSIS

```
int dbi_dbms(dbms_statement)
char * dbms_statement;
```

DESCRIPTION

Parses, validates and executes a DBMS statement. The DBMS statement must be syntactically the same as a JPL DBMS statement, *but without the JPL verb DBMS at the beginning.*

RETURNS

0 if no error
-1 if an error.

NAME

dbi_sql -parse and execute a SQL statement

SYNOPSIS

```
int dbi_sql(sql_statement)
char * sql_statement;
```

DESCRIPTION

Parses, validates and executes a SQL statement. The SQL statement must be syntactically the same as a JPL SQL statement, *but without the JPL verb SQL at the beginning*. The statement must also be syntactically correct for the database being used.

NOTE: Because no JPL parsing is done on SQL statements called by this function, there is no colon substitution of variables.

RETURNS

0 if no error
-1 if an error.

---

3 Initialization

Most databases require a logon procedure or a function call. The actual parameters used in the logon depend on the database. See Appendix B for details.

The syntax of the logon command is:

```
DBMS LOGON <other arguments>
```

If, for example, one were using Gupta Technologies' SQLBase and the data dictionary variables DBNAME (for database name), USER (for the user name) and PASSWORD (for the user's password), the logon command would look like this:

```
DBMS LOGON :DBNAME :USER :PASSWORD
```

The phrases :DBNAME, :USER and :PASSWORD are variables which JAM/DB*i* will replace with correct values from the JAM data dictionary. The process of variable substitution is described below.

The obverse of the logon command is *logoff*. There are no arguments to the logoff command:

```
DBMS LOGOFF
```

You should always execute a LOGOFF to disconnect properly. Otherwise, you may create inconsistencies in your database.

4 Fetching and Inserting Data

JAM/DB*i* permits you to move data between a JAM application and a supported database. JAM/DB*i* may insert data into, or update data from, a JAM screen, a JAM data dictionary and/or a supported database.

All data manipulation in JAM/DB*i* is done using the JPL verb SQL. The SQL verb in turn expects to be passed a SQL data manipulation string using column-based instructions such as SELECT, INSERT, UPDATE and DELETE, or table- or view-based instructions such as CREATE, ALTER or DROP.

JAM/DB*i* manipulates all SQL statements dynamically, creating temporary data storage space, doing any requisite data conversions and, in the case of a SELECT, moving data into a JAM screen or the JAM data dictionary.

JAM/DB*i* pre-parses your SQL statement and makes any necessary variable substitutions. JAM/DB*i* then passes on to your database system the SQL statement you asked JAM/DB*i* to prepare.

All database errors are trapped and will be displayed (depending on the environmental controls set by the developer; see Section 5 below). Warnings may be ignored, depending on the environmental handling set by the application developer.

IMPORTANT DEPENDENCY: JAM/DBi SQL syntax is native to the database you are using. For example, some databases convert column names to lower (upper) case and return data with the lower case mapping. Users who define JAM screen or data dictionary names in upper (lower) case with such databases will not find any data being passed back and forth between JAM and the database.

4.1 Variable Substitution

A SQL statement such as

```
SQL SELECT NAME, ADDRESS FROM EMP
```

will always search and return all instances of data with name and address in the table EMP. In many cases, however, such a SQL statement needs to be qualified. For example:

will always search and return all instances of data with name and address in the table EMP. In many cases, however, such a SQL statement needs to be qualified. For example:

```
SQL SELECT NAME, ADDRESS FROM EMP WHERE\
                        NAME='JOHN'
```

In this case, SQL will search for and return all instances of data where name is equal to JOHN.

For most applications, the qualifying value in the WHERE clause of a SQL statement is not known until runtime. To handle such situations, JAM/DB*i* allows dynamic variable substitution in SQL statements.

Substitution variables are variables in a SQL statement that are replaced with values from the JAM application screen or the data dictionary. These correspond to standard SQL host variables. However, JAM/DB*i* provides an extended capability in that substitution variables can contain any character string for substitution. This means that substitution variables may contain whole or partial SQL statements for substitution (see below).

4.1.1 Defining Substitution Variables

A substitution variable is identified by a preceding colon (e.g. :MYVAR). Colons in the middle of a word (e.g. MY:VAR) will be ignored. When a colon is detected, the word following it is used to search the field list on the JAM screen and then the JAM data dictionary. (A double colon (::) can be used to suppress substitution.)

If the substitution variable name is found, the corresponding value is inserted into the SQL statement in place of the substitution variable name.

The substitution variable may be a single element, a complete array or an element of one. If an array name is specified without reference to a specific array element, all non-blank fields in the array are inserted in place of the substitution variable. The inserted fields of a complete array are separated by single spaces.

Examples of using substitution variables follow:

```
SQL INSERT INTO EMP VALUES\
(':NEWNAME',:SALARY,':START_DATE',:EMPLOY_NUM,'HIRE')
```

```
SQL UPDATE EMP SET SALARY=:SALARY_TABLE[4]\
WHERE EMPLOY_NUM=:EMPLOY_NUM
SQL :SQL_STATEMENT
```

The first SQL statement adds a new row of data to the database table EMP with the JAM string variables :NEWNAME and :START_DATE, the JAM numeric variables :SALARY and :EMPLOY_NUM and a fixed string HIRE.

The second statement modifies an existing row or rows of data in the table EMP. It changes all instances of SALARY in EMP to the value of element 4 of a JAM array field called SALARY_TABLE. In that array, the column EMPLOY_NUM is equal to JAM data field EMPLOY_NUM.

The third SQL statement replaces the JAM/DB*i* SQL variable :SQL_STATEMENT with whatever is found in the JAM data field SQL_STATEMENT. In this case, presumably, SQL_STATEMENT would hold an entire SQL statement. Users may also use nested bindings with JPL. See JAM JPL Programmer's Guide for instructions.

Note that SQL statements in JPL are not terminated by a semi-colon (;).

4.2 Column Mapping and Aliases

When a SELECT statement is executed, JAM/DB*i* copies returned values into JAM screen or data dictionary fields, if any. For JAM/DB*i* to do this, there *must* be a one-to-one mapping between SQL column names and JAM field names. For example, a SQL statement retrieving data from EMPLOY_NAME will attempt to place the returned data into a JAM field of exactly the same name. (Names are truncated in JAM to 31 characters.) If such a field is not found, JAM/DB*i* will ignore the returned value for that column.

It is because of this mapping that users can invoke a SQL statement like the following:

```
SQL  SELECT * FROM EMP
```

However, there are times when a one-to-one mapping is not possible or it is too constraining. In such cases, many databases permit a remapping of names. For example, if the database column was named EMP_NAME and the JAM field was named EMPLOYEE_NAME (perhaps because EMP_NAME was already used for some other purpose), the developer can remap the association of database and JAM field names. This is done within the SQL statement itself. Using our example where the database column name is EMP_NAME, the JAM fields EMP_NAME and EMPLOYEE_NAME are defined and the developer does not want to change the current JAM field value of EMP_NAME, the appropriate SQL statement would be:

```
SQL SELECT EMP_NAME EMPLOYEE_NAME
FROM EMP WHERE EMPLOY_NUM=:EMPLOY_NUM
```

Note that there is *no comma* between EMP_NAME and EMPLOYEE_NAME. It is the absence of the comma that permits the parser to map the association of the column name EMP_NAME with the JAM field name of EMPLOYEE_NAME. A comma between the two names would have caused a SQL SELECT error if EMPLOYEE_NAME were not also in the table EMP.

IMPORTANT DEPENDENCY: There are some databases that DO NOT SUPPORT COLUMN REMAPPING. Check in Appendix B to determine whether your database supports column remapping. If your database does not support remapping, JAM/DB*i* provides an alternative means of accomplishing the same thing using a DBMS command called DBMS SELECT_ALIAS.

SELECT_ALIAS is available *only* in those databases that do support remapping. A SELECT_ALIAS must be executed just before the SQL SELECT statement to be parsed. SELECT_ALIAS must be used if *any* of the column names in the SQL SELECT statements do not directly correspond to the JAM field names (e.g., if the DBMS column name is NAME and the JAM field name is CLIENT_NAME). An example of SELECT_ALIAS is:

```
DBMS SELECT_ALIAS CLIENT_NAME, TEST, -, RESULT
```

for the SQL statement:

```
SQL SELECT NAME, GRADE, AGE, SCORE FROM XYZ
```

The hyphen in the SELECT_ALIAS means that no remapping is required for the third column name in the SQL SELECT (i.e., AGE). There is a one-to-one correspondence between the number of arguments in SELECT_ALIAS and the number of columns specified in a SQL SELECT.

4.3 Fetching

Most of the information you need to fetch data with SELECT has been specified above. However, there are several implementation details that are important.

1. When retrieving multiple rows of data, JAM/DB*i* will determine the maximum number of rows that can be retrieved at one time. In the event a JAM field is defined as an array in the screen or the data dictionary, JAM/DB*i* will take the minimum number of defined occurrences of the field in the screen or the data dictionary. While developers are strongly discouraged from creating arrays in a form and the data dictionary of the same name but different size (measured in terms of array element occurrences), JAM/DB*i* will protect the developer by returning the lesser of the occurrences. To continue retrieving data, see the DBMS CONTINUE command below.
2. The maximum number of rows of data that JAM/DB*i* will return in a single fetch is based on the smallest number of array occurrences of any single data element in the fetch. As an example, if you execute SQL SELECT NAME, ADDRESS, STATE FROM EMP, and NAME and ADDRESS have been defined in a JAM application as arrays of 15 occurences each and STATE as an array of 10 occurrences, then the maximum number of rows returned will be 10 at a time.
3. Some SQL developers may be used to forcing the closing of a SQL data storage area (called a SQL cursor). In JAM/DB*i*, you do not have to force the closing of a SQL cursor. JAM/DB*i* will automatically open and close cursors.
4. Because JAM Version 4.0 permits fields to be defined on a screen and not necessarily in a data dictionary, JAM/DB*i* uses the following order of precedence when searching for a field name and its characteristics:

a. Screen variables
b. Data dictionary variables

5. If after searching these lists JAM/DB*i* cannot match a field name with a SQL column name, JAM/DB*i* will ignore that SQL column in the subsequent retrieval of data.
6. JAM fields may be defined anywhere. Subsequently, a SQL statement may call for the retrieval of data where some fields are defined only in the JAM screen, where others are defined only in the JAM data dictionary and yet others are defined in both places.
7. Some users of JAM/DB*i* Version 3.X will find that some JAM/JPL DBMS verbs are no longer supported (DIAG in Informix and AUTOCOMMIT in ORACLE).

4.4 Continuing

If a SELECT returns more rows than can be placed into a JAM field array (wherever it is defined), you can subsequently continue to retrieve more data by attaching a DBMS CONTINUE statement in a JPL file.

Moreover, if a user explicitly calls a CONTINUE after SQL has indicated there are no more rows to fetch, JAM/DB*i* will not access the database. To control this, the developer should set the environment variable ERROR (see Section 5) and constantly check the current value of the ERROR variable for a 'no more rows' condition. The actual value returned is database dependent.

Alternatively, users can use the DBMS COUNT function to check the number of rows returned by the SQL SELECT and maintain their own current count of how many rows are left in the fetch. (This is obviously more problematic for databases that permit forward and backward retrieval of rows.)

4.5 Next/Cancel/Flush

4.5.1 Next

Some databases support multiple commands to be issued in a single string. If your database supports this feature, JAM/DB*i* will attempt to process such strings. If, however, a command (other than the last command in a sequence) returns data (via

SELECT) and you are required to issue a DBMS CONTINUE, the original command will be suspended. To continue processing a multicommand string from a suspended state, issue the following:

```
DBMS NEXT
```

4.5.2 Cancel

If there are multiple commands in a single string and processing has been suspended as described above, the remaining commands may be canceled with the following command:

```
DBMS CANCEL
```

4.5.3 Flushing

Some database systems using multicommand strings (e.g., Britton Lee) require that, when a SELECT has returned multiple rows and not all rows are fetched by JAM/DB*i*, the application must explicitly flush unread rows before another command is processed. Since there are no side effects to flushing, you may issue a flush command even if you are not sure whether there are any more rows left. The syntax for this command is:

```
DBMS FLUSH
```

5 JAM/DBi Environment

JAM/DB*i* provides several functions to control processing, error trapping and database maintenance. These functions frequently use native functions provided by the database system.

5.1 Determining the Number of Rows a SELECT Will Find

Since JAM/DB*i* keeps track of only the rows read but not of how many are held in a cursor (some databases do not easily provide that information), you can find out how many records will be found by a given SQL SELECT by issuing the SELECT COUNT command, as explained below.

First create a variable in the JAM data dictionary to hold the return value. Let's assume we have such a variable, called TOTAL.

```
SQL SELECT COUNT (EMPLOY_NUM) TOTAL\
FROM EMP WHERE EMPLOY_NUM > :EMPLOY_NUM
```

A subsequent SQL statement that uses the same table and WHERE clause will return the same number of rows as the number in TOTAL.

5.2 Determining the Number of Returned (Read) Rows

When a SQL SELECT is issued, there is no guarantee of how many rows of data the database will return. Frequently, it is important to know either that no rows were returned or how many rows must be processed.

Note that the DBMS COUNT function does not return the total number of rows found by SQL for a specific SELECT. (Use a SQL COUNT to do that.) It returns the number of rows read into memory from the current SQL cursor.

To find out how many rows were returned, define either a JAM screen field or data dictionary field to hold the row count (e.g. MYCOUNT), and issue the following command:

```
DBMS COUNT MYCOUNT
```

After each SELECT and any subsequent CONTINUE, JAM/DB*i* will place the returned row count into MYCOUNT.

DBMS COUNT can also be used for SQL DELETE and SQL UPDATE.

5.3 Specifying a Start Row

Some databases do not offer any means to page backwards through the database, nor does the current version of JAM/DB*i* provide any direct support for backward paging (or redirected start and end). To help you around this problem, JAM/DB*i* provides a means for reading a predetermined number of records and discarding them before beginning to read the records that you want to see.

Let's assume that you want to page backwards through the database. Just issue the following commands:

```
VARS RUNNING_COUNT
CAT RUNNING_COUNT "0"
DBMS START :RUNNING_COUNT
DBMS COUNT MYCOUNT
RETURN 0
```

Issue a SQL SELECT COUNT command:

```
SQL SELECT COUNT (EMPLOY_NUM) TOTAL FROM EMP
```

This number can be used to make sure that you do not create a starting row value greater than the number of rows that can be returned.

Then issue your SELECT statement:

```
SQL SELECT EMPLOY_NAME,EMPLOY_NUM FROM EMP
```

After each SELECT or CONTINUE, add the value in MYCOUNT to RUNNING_COUNT.

Let us also assume that you have an array that holds 15 rows of data and that the SQL statement has returned the value of 50 into TOTAL. Furthermore, assume that you have read 3 pages (or 45 rows of data), of which the first 2 pages are lost. To go back one page of data, issue the following JAM/JPL command:

```
MATH RUNNING_COUNT=RUNNING_COUNT - 30
IF RUNNING_COUNT <"0"
{
CAT RUNNING_COUNT "0"
}
DBMS START:RUNNING_COUNT
```

```
SQL SELECT EMPLOY_NAME,EMPLOY_NUM FROM EMP
RETURN 0
```

Of course, if rows are added to or deleted from the table by other users, the starting point will be only proximate to the top of each page as the user pages forwards.

5.4 Error Processing

By default, JAM/DB*i* displays any database errors at the bottom of the screen. If you want to control error processing yourself, use the DBMS ERROR command to define a field to hold the database return code and, optionally, an error message. For example, to save the return value of a database status, create a field called MYERROR and issue the following:

```
DBMS ERROR MYERROR
```

If you want to store a return error message from the database, you can specify a data dictionary element (and optionally an occurrence). For example, if you have an array ERROR_MSG with 20 occurrences and you want to store a database error in the 8th occurrence, use the following statement:

```
DBMS ERROR MYERROR ERROR_MSGS[8]
```

To return to the default condition, just issue the following:

```
DBMS ERROR
```

Users should note that in JAM/DB*i* Version 4.0 SQL errors will not terminate JPL processing. In Version 3.X, on the other hand, if error trapping was turned off, SQL errors would stop JPL processing.

Note: The DBMS ERROR function, if active, passes on only those errors that are returned by the database. Errors in JPL syntax will report an error and terminate the JPL procedure. Moreover, some databases do not return error messages, and in some cases users are required to fetch messages from a database file on the basis of an error number.

5.5 Warning Processing

By default, JAM/DB*i* ignores database warnings. To catch and process warnings, first create a JAM data dictionary variable (for example, MYWARNING) to store the warning message and then issue the command:

```
DBMS WARN MYWARNING
```

You can also use JAM arrays. Note that warning formats vary from database to database. See Appendix B for details. You can return to the default status by issuing the following:

```
DBMS WARN
```

5.6 Begin/Commit/Rollback

If your database supports "before image journaling", you may create sets of transactions that may be written to the database at once. Such sets provide the opportunity to "rollback" an entire set of interrelated transactions if one of the SQL transactions fails. The way this procedure works varies from database to database. The basic process includes marking the beginning of a transaction, executing one or more transactions and then, after testing the return codes of these transactions, executing either a COMMIT or ROLLBACK. In some databases, there is no need to mark the beginning of a transaction since all transactions since the last COMMIT or ROLLBACK will be affected by a COMMIT or ROLLBACK.

The syntax for these commands is:

```
DBMS BEGIN
DBMS COMMIT
DBMS ROLLBACK
```

6 JAM/DBi Utilities

Version 4.0 of JAM/DB*i* provides two utilities to aid in program development. The first utility - `f2tbl` - creates a database table from a binary JAM form. The second utility - `tbl2f` - creates a basic JAM form from a database table definition.

NAME

`f2tbl` - form to table utility

SYNOPSIS

`f2tbl <JAM file name>`

DESCRIPTION

This utility program will take a user-specified JAM form in binary file format, identify JAM fields and data types and create a table in the user's database with those fields.

This version of `f2tbl` maps a one-to-one correspondence between form and table. Future versions will permit a form to map to different tables.

The user is prompted for the name of the table to be created. If the database being used requires a database name, user name and/or password, the user will also be prompted for that information.

`f2tbl` will exclude JAM control fields (i.e., those defined with a `jam_` definition and fields that are not named).

`f2tbl` will create a maximum of 50 columns (JAM fields) in a table.

The only two data types supported are character and integer. All other data types will be converted to one of these types.

NAME

tbl2f - table to form utility

SYNOPSIS

```
tbl2f
```

DESCRIPTION

This utility program will take a database table and create a binary JAM form with named fields corresponding to the fields in the database table.

This version of tbl2f maps a one-to-one correspondence between table and form. Future versions will permit multiple tables to map to a single form.

If the database requires a database name, user name and/or password, the user is prompted for such information. tbl2f will prompt the user for a table name. tbl2f will then prompt the user for a form name. The default is the database table truncated (if necessary) to a valid file name length with the extension .JAM.

A maximum of 24 fields will be created. tbl2f will inform the user of how many fields were created or how many were dropped from the specification if the number of columns exceeds 24.

The three data types supported are character, integer and float. All other data types will be converted to one of these types. Note that LONG VARS (variable length) will be created but data will be truncated. Future versions of JAM/DB*i* will support LONG and RAW vars.

Each field will have a maximum on-screen size of 20, but fields larger than 20 characters will be made into shifting fields.