# **JAM**/DB*i*

## Release 5

## Document Structure

The JAM/DB*i* Manual is printed in one volume. It is divided into the following sections:

- JAM/DB*i* Overview – An overview of the JAM/DB*i* product and the development process. This section describes JAM/DB*i* from the "big picture" viewpoint. It describes all the pieces of a sample JAM/DB*i* application.
- Developer's Guide – A guide to using JAM/DB*i* features. This section is divided into four main sections:—accessing a database with JAM/DB*i* structures, sending JAM values to a database, sending database values to JAM variables, and using transactions.
- Reference Guide – Manual pages for the `dbms` commands, the JAM/DB*i* library functions, and the JAM/DB*i* utilities.
- Notes – A description of features and discussion of topics particular to an engine.
- Appendices – These include lists of keywords, error codes, and suggestions on using JAM more effectively with a JAM/DB*i* application.
- Index

## Terminology

Terms will be defined when discussed. However, it is necessary to define a few that will be used throughout the manual.

- *database* — A physical database consisting of tables and other data.
- *vendor* — A supplier of a DBMS engine.
- *engine* — A DBMS product. An engine is identified by a specific vendor and version.

## Notation

To make this manual easier to use, we use the notation described below.

- `literal` — We use this font for text that you will type verbatim. In particular, we use this font for all examples. We also use it when naming a JAM library function, a JPL command, or a utility.
- SMALL CAPS — It is customary to put SQL keywords in uppercase. We follow this convention. In addition, in synopses of `dbms` com-

mands, we put `dbms` keywords in uppercase. Please note that the use of case is purely stylistic. Case is significant only for identifiers—names of fields, columns, tables, variables. functions, etc.

- ***italics*** We use bold italics to show where variable or procedure names should appear. Text in this font should be replaced with a specific, appropriate value in an application.
- [***x***] Brackets indicate an optional element. The brackets should not be typed.
- ***x***... Ellipses indicate that an element may be repeated one or more times.

# TABLE OF CONTENTS


I. JAM/DBi Overview .................................. 1

Chapter 1.
Introduction .......................................... 1

Chapter 2.
What is JAM/DBi? ...................................... 2

2.1. Components of JAM/DBi Architecture ................ 3
2.2. Components of JAM/DBi ............................. 5
2.2.1. JAM/DBi Libraries ................................ 5
2.2.2. Source Code ...................................... 5
2.2.3. Header Files ..................................... 6
2.2.4. Makefile ......................................... 6
2.3. Components of a JAM/DBi Application ............... 6

Chapter 3.
JAM/DBi Application Development ...................... 9

3.1. Creating and Editing Application Screens .......... 9
3.1.1. Mapping Columns to JAM Variables ................. 10
Automatic Mapping ....................................... 10
Aliasing ................................................ 10
3.1.2. Data Validation .................................. 11
3.2. Error Handling .................................... 11
3.3. Iterative Application Testing ..................... 12

Chapter 4.
JAM/DBi Control Flow ................................. 13

4.1. Sample Application – User's View .................. 14
4.2. Sample Application – Developer's View ............. 16
4.2.1. Database Tables `emp`, `acc`, and `review` ....... 17
4.2.2. Source Module `dbiinit.c` ........................ 19
4.2.3. Data Dictionary and Initialization File .......... 21
4.2.4. JAM Screens ...................................... 22
Main Screen ............................................. 22

Employee Screen .......... 26
Salary History Screen .......... 33
4.3. JAM/DBi Control Flow SUMMARY .......... 35
4.3.1. Variable Substitution .......... 36
4.3.2. Cursors .......... 36
Fetching a SELECT Set Incrementally .......... 37
Using Multiple SELECT Sets .......... 37
Improving Efficiency .......... 38
4.3.3. Error Processing .......... 38

**Chapter 5.**
**JAM/DBi Philosophy .......... 40**
5.1. JAM/DBi Features .......... 40
5.1.1. SQL-Based .......... 40
5.1.2. OS Portability .......... 41
5.1.3. Vendor Independence .......... 41
5.1.4. Multi-engine Support .......... 41
5.1.5. Multi-connection Support .......... 42
5.1.6. Prototyping .......... 43
5.2. JAM/DBi Development Hints .......... 44

**II. Developer's Guide .......... 45**

**Chapter 6.**
**Introduction to Development .......... 47**
6.1. SQL Variants .......... 47
6.2. JAM/DBi Commands .......... 48
6.2.1. JPL versus C .......... 49

**Chapter 7.**
**Access and Execution .......... 51**
7.1. Initializing One or More Engines .......... 52
7.1.1. Initializing an Engine in `dbiinit.c` .......... 52
7.1.2. Initialization Procedure .......... 54
7.1.3. Setting the Default Engine .......... 54

7.2. Connecting to a Database Server .................................... 55
7.2.1. Connections to Multiple Engines ................................ 55
7.2.2. Multiple Connections to a Single Engine ........................ 56
7.3. Using Cursors ..................................................... 57
7.3.1. Using the Default Cursor ....................................... 57
7.3.2. Using a Named Cursor ........................................... 58
Declaring a Cursor ................................................ 58
Executing a Cursor ................................................ 59
Modifying or Closing a Cursor ..................................... 60

**Chapter 8.**
**Data Flow from JAM .................................. 61**

8.1. Colon Preprocessing ............................................... 62
8.1.1. Colon-plus Processing .......................................... 62
Step 1. Perform Standard Colon Preprocessing ...................... 64
Step 2. Determine the Variable's JAM Type ......................... 64
Step 3. Format a Non-null Value ................................... 66
8.1.2. Colon-equal Processing ......................................... 68
8.1.3. Examples ....................................................... 69
A Field with Default Characteristics .............................. 69
A Variable with a Date-time Edit and a Null Edit .................. 70
A Variable with a Digits Only Character Edit and a C-Type Edit ..... 71
8.2. Using Parameters in a Cursor Declaration .......................... 72
8.2.1. Parameter Substitution and Formatting .......................... 73
Examples .......................................................... 74

**Chapter 9.**
**Data Flow from a Database ............................ 77**

9.1. Data Fetched by SELECT ............................................ 78
9.1.1. JAM Targets for a SELECT ....................................... 78
Automatic Mapping ................................................. 79
Aliasing .......................................................... 79
9.1.2. Number of Rows Fetched ......................................... 83
Scrolling Through a select Set .................................... 83
Controlling the Number of Rows Fetched ............................ 88
Choosing a Starting Row in the select Set ......................... 88
9.1.3. Format of select Results ....................................... 89

Character Column .......... 89
Date-time Column .......... 89
Numeric Column .......... 89
Fetching Unique Column Values .......... 90
9.1.4. Redirecting select Results to Other Targets .......... 92
9.2. Status and Error Codes .......... 93

**Chapter 10.**
**Hook Functions .......... 95**
10.1. ONENTRY Function .......... 96
10.1.1. ONENTRY Function Arguments .......... 96
10.1.2. ONENTRY Return Codes .......... 96
10.1.3. Example ONENTRY Functions .......... 97
10.2. ONEXIT function .......... 98
10.2.1. ONEXIT Function Arguments .......... 98
10.2.2. ONEXIT Return Codes .......... 98
10.2.3. Example ONEXIT Function .......... 98
10.3. ONERROR Function .......... 99
10.3.1. ONERROR Function Arguments .......... 100
10.3.2. ONERROR Return Codes .......... 101
10.3.3. Example ONERROR Function .......... 101

**Chapter 11.**
**Transactions .......... 103**
11.1. Engine-specific Behavior .......... 104
11.2. Error Processing for a Transaction .......... 105

**III. Reference Guide .......... 109**

**Chapter 12.**
**JAM/DBi Reference Overview .......... 111**

**Chapter 13.**
**DBMS Global Variables .......... 113**

13.1. Variable Overview ........ 113
13.1.1. Error Data ........ 113
13.1.2. Status Data ........ 114
13.2. Variable Reference ........ 114
@dmengerrcode contains an engine-specific error code ........ 115
@dmengerrmsg contains an engine-specific error message ........ 117
@dmengreturn contains a return code from a stored procedure ........ 118
@dmengwarncode contains an engine-specific warning code ........ 120
@dmengwarnmsg contains an engine-specific warning message ........ 121
@dmretcode contains an engine-independent error or status code ........ 122
@dmretmsg contains an engine-independent error or status message ........ 124
@dmrowcount contains a count of the number of rows fetched to JAM ........ 125
@dmserial contains a serial column value after performing INSERT ........ 127

## Chapter 14. DBMS Commands ........ 129

14.1. DBMS Command Overview ........ 129
14.2. Command Reference ........ 131
ALIAS set aliases for a declared or default SELECT cursor ........ 133
BINARY define JAM/DBi variables for fetching binary values ........ 136
CATQUERY concatenate a full result row to a JAM variable or a file ........ 138
CLOSE_ALL_CONNECTIONS close all connections on an engine ........ 141
CLOSE CONNECTION close a declared connection ........ 142
CLOSE CURSOR close a named or default cursor ........ 143
CONNECTION set or change the default connection ........ 145
CONTINUE fetch the next set of rows associated with a SELECT cursor ........ 146
CONTINUE_BOTTOM fetch the last page of rows associated with a SELECT cursor ........ 148
CONTINUE _DOWN fetch the next set of rows associated with a SELECT cursor ........ 149
CONTINUE_TOP fetch the first page of rows associated with a SELECT cursor ........ 150
CONTINUE_UP fetch the previous page of rows associated a SELECT cursor ........ 152
DECLARE CONNECTION create a named connection to a server and database ........ 154
DECLARE CURSOR declare a named cursor for a SQL statement ........ 155
ENGINE set or change the default engine ........ 157
EXECUTE execute the SQL statement declared for a named cursor ........ 158
FORMAT format catquery values ........ 160
OCCUR change the behavior of a select cursor that writes to JAM arrays 162
ONENTRY install an entry function ........ 164
ONERROR set the behavior of the error handler ........ 166

| | | |
|---|---|---|
| ONEXIT | install an exit handler | 168 |
| START | specify a starting row in a SELECT set | 170 |
| STORE | set up a continuation file for a named or default cursor | 171 |
| UNIQUE | suppress repeating values in selected columns | 174 |
| WITH CONNECTION | use a named connection for the duration of a statement | 175 |
| WITH CURSOR | use a named cursor for the duration of a statement | 177 |
| WITH ENGINE | use a named engine for the duration of a statement | 179 |

## Chapter 15. JAM/DBi Library Reference ... 181

| | | |
|---|---|---|
| dm_bin_create_occur | get or allocate an occurrence in a binary variable | 183 |
| dm_bin_delete_occur | delete an occurrence in a binary variable | 184 |
| dm_bin_get_dlength | get the length of an occurrence in a binary variable | 185 |
| dm_bin_get_occur | get the data in an occurrence of a binary variable | 186 |
| dm_bin_length | get the maximum length of an occurrence in a binary variable | 187 |
| dm_bin_max_occur | get the maximum number of occurrences in a binary variable | 188 |
| dm_bin_set_dlength | set the length of an occurrence in a binary variable | 189 |
| dm_dbi_init | initialize JAM for JAM/DBi | 190 |
| dm_dbms | execute a DBMS command directly from C | 191 |
| dm_dbms_noexp | execute a DBMS command without colon preprocessing | 193 |
| dm_expand | format a string for an engine | 194 |
| dm_getdbitext | get the text of the last executed dbms or sql command | 198 |
| dm_init | initialize JAM/DBi to access a specific database engine | 199 |
| dm_reset | disable support for a named engine | 201 |
| dm_sql | execute a SQL command directly from C | 202 |
| dm_sql_noexp | execute a SQL command without colon preprocessing | 203 |

## Chapter 16. JAM/DBi Utility Reference ... 205

| | | |
|---|---|---|
| f2tbl | create a database table from a JAM form | 206 |
| tbl2f | create a JAM screen from a database table | 213 |

# V. Appendixes ... 239

Appendix A. Keywords ... A-1

Appendix B. Error and Status Codes ... B-1

Appendix C.
Using the JAM Screen Editor .................................... C-1

# JAM/DB*i*
# Overview

# Chapter 1.
# *Introduction*

This document is intended for developers who are using JAM/DB*i*® for the first time, or for those who wish to gain a better understanding of this product. This document is intended to provide a conceptual overview of JAM/DB*i*. It will help you understand and use JAM/DB*i*.

JAM/DB*i* is part of a family of JYACC products. The following table describes the rest of the family:

| *Product* | *Description* |
|---|---|
| JAM® | JYACC Application Manager |
| JAM/DB*i* Report writer | Report Writer for JAM/DB*i* |
| JAM/P*i* for Motif | Presentation interface for the Motif GUI |
| JAM/P*i* for Microsoft Windows | Presentation interface for Microsoft Windows |
| JAM/P*i* for Graphics | Presentation interface for Graphics |
| *Jterm*® | Color Terminal Emulator optimized for JAM |

**If you are upgrading from Release 4.8, please read Chapter 21. "Summary of New Features" and Chapter 22. "Release 4.8 Compatibility."**

# Chapter 2.
# *What is* JAM/DB*i*?

JAM is a software toolkit that aids developers in prototyping and building applications with sophisticated interfaces. JAM provides tools for creating screens that accept and display data for end users, and that define the control flow of an application.

JAM/DB*i* is a portable interface between JAM applications and relational database systems. It provides facilities for the gamut of data manipulation needs. In particular, a developer may build a JAM/DB*i* application which permits end users to perform any of the following:

- Retrieve values from database tables for display on screens. Queries may be hard-coded, or they be created at runtime according to an end user's specifications.
- Add rows to or delete rows from database tables.
- Update existing rows.
- Create or drop database tables.
- Execute any other function provided by the database's dynamic query interface (e.g., execute a stored procedure, rollback a transaction, etc.).

The rest of this document assumes that you are familiar with JAM and the concepts discussed in the JAM Overview. In addition, it assumes that you have some database experience.

# 2.1. COMPONENTS OF JAM/DB*i* ARCHITECTURE

There are several layers in the JAM/DB*i* architecture.

1. JAM Application – This typically includes the following:
   - Menu screens for control flow in the application;
   - Screens for entering new values to a database;
   - Screens for viewing and updating information in a database;
   - Related hook functions.
2. JAM/DB*i* – The interface between a JAM application and a DBMS client library. The interface has a generic part and one or more specific parts called "support routines." A support routine provides access to a particular DBMS product, also called an "engine."
3. DBMS Client Library – The interface that controls all programmed access to a database. This is the interface between JAM/DB*i* and a DBMS. The DBMS controls all access to the database.
4. DBMS Network Services – The network services that connect a user's client library with one or more DBMS servers.
5. DBMS Server.

See the figure below.

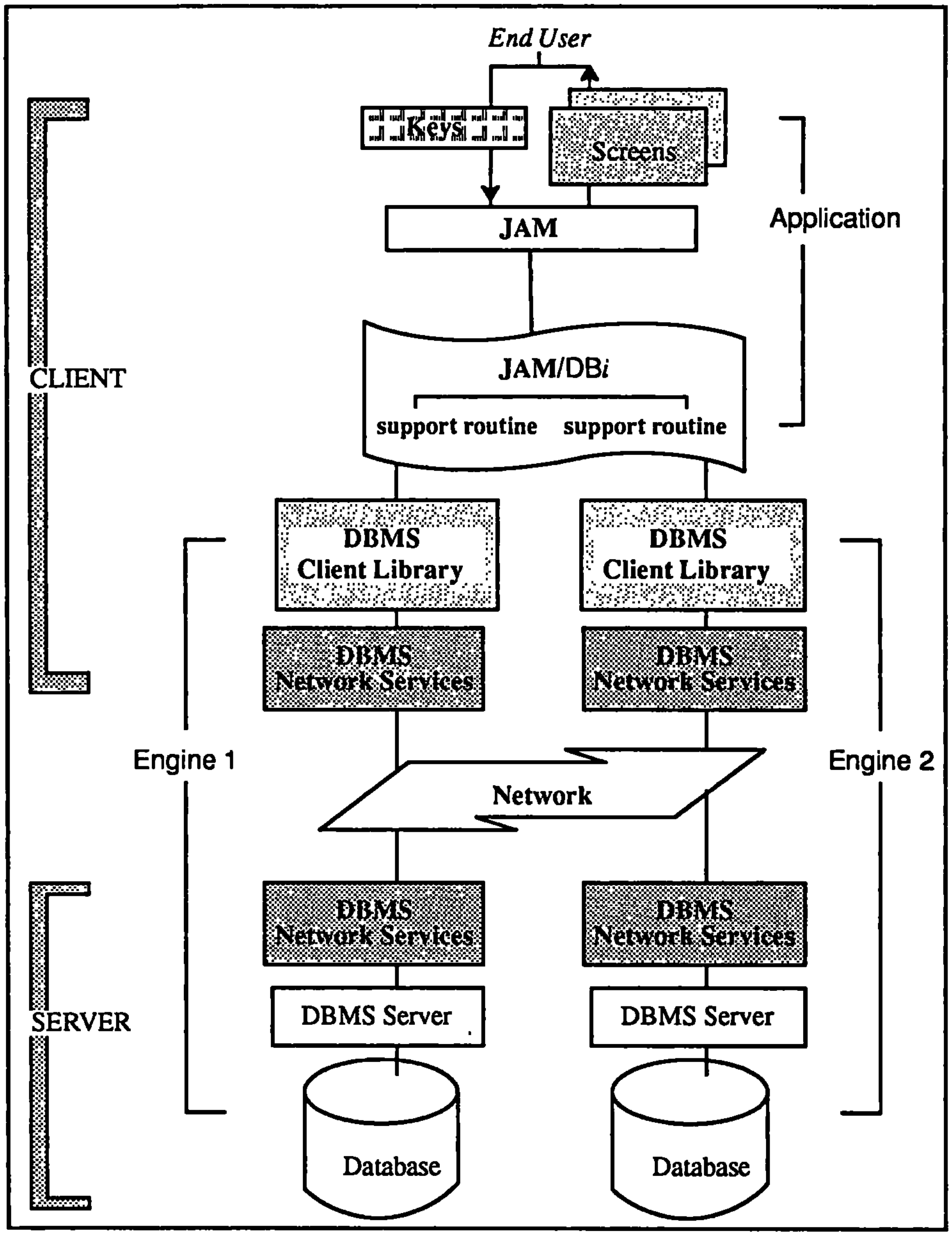



Figure 1: Components of JAM/DB*i* Architecture.

## 2.2. COMPONENTS OF JAM/DB*i*

The JAM/DB*i* product is collection of programs and data files. In the sections below, we briefly discuss the main components of the product. For more information, see the README file included with the distribution.

### 2.2.1. JAM/DB*i* Libraries

The JAM/DB*i* interface is written using tools provided by your database vendor, either embedded SQL or a C language API (applications programming interface). A JAM/DB*i* developer does not need to write any code using embedded SQL or an API, but in order to link an application he or she must have access the header files and libraries supplied with these tools. The README provided with the JAM/DB*i* distribution names and describes the necessary products.

Each JAM/DB*i* supplies a "common" library and one or more engine-specific libraries. The additional engine-specific libraries are provided so that JAM/DB*i* may support different versions of a database, or support different modes, for example on MSDOS, real mode and Windows mode. The library names are database-specific, usually in the form `lib`***db***`.a` or `llib`***db***`.lib` with ***db*** representing a vendor name. For example, ***db*** may be `ora` for ORACLE or `syb` for SYBASE.The JAM/DB*i* README file names and describes the libraries for your database.

### 2.2.2. Source Code

The JAM/DB*i* source code module is `dbiinit.c`. Customized for a particular engine, it specifies header files needed by JAM/DB*i*, declares the name of the support routine for the engine, and sets up some defaults for handling errors and case-sensitivity.

2.2.3.

### Header Files

JAM/DB*i* supplies some header files. The file `dmerror.h` defines symbolic constants and integer codes for JAM/DB*i* and DBMS errors. The README file provides a complete list of the distribution header files.

2.2.4.

### Makefile

Once you have edited the makefile to describe the engine version and the pathname to its installation, you must run the makefile to create the JAM/DB*i* executables, `jamdbi`, `jxdbi`, `f2tbl`, and `tbl2f`. See the installation notes and instructions in the makefile for more information.

2.3.

## COMPONENTS OF A JAM/DB*i* APPLICATION

New users are sometimes confused about the differences between JAM applications and JAM/DB*i* applications. They share many similarities, as shown in the table below.

| JAM Application | JAM/DB*i* Application |
|---|---|
| JAM Screens | JAM screens |
| Data Dictionary | Data Dictionary |
| Hook Functions (JPL and/or C) | Hook Functions (JPL and/or C); Hook functions include database function calls |
| JAM Executable | JAM/DB*i* Executable |

In a JAM/DB*i* application, you can log on, query, or update a database. These functions cannot be performed in a standard JAM application unless you write your own database interface.

If you are familiar with JAM, you are familiar with the two types of JAM executables—the authoring executable and the application executable. (If not, see the introductory chapters of the JAM Programmer's Guide.) Similarly, JAM/DB*i* has two executable versions—the authoring executable, sometimes called `jxdbi`, and the application executable, sometimes called `jamdbi`. The authoring executable links the developer's hook functions with the **JAM** Screen Manager, **JAM** Executive, and authoring libraries, as well as the **JAM/DB*i*** interface libraries and the engine's libraries. It is used to create and test an application. The application executable, on the other hand, is a runtime program which you may distribute to end users. It does not provide access to the **JAM** Screen, Keyset, or Data Dictionary Editors.

The tables below list and compare the files which developers must link when creating the executables. We describe the **JAM/DB*i*** files at the end of the section.

| **JAM *Authoring* Executable** | **JAM/DB*i* *Authoring* Executable** |
|---|---|
| `jxmain.o` | `jxmain.o` |
| `funclist.o` | `funclist.o` |
| | `dbiinit.o` |
| **JAM** Authoring Library (JX) | **JAM** Authoring Library (JX) |
| **JAM** Executive Library (JM) | **JAM** Executive Library (JM) |
| **JAM** Screen Manager Library (SM) | **JAM** Screen Manager Library (SM) |
| | **JAM/DB*i*** Common Interface Library (DM) |
| | **JAM/DB*i*** Engine-specific Interface Library (1 or more for each DBMS) |
| | DBMS Client Library (1 or more for each DBMS) |

| **JAM *Application* Executable** | **JAM/DB*i* *Application* Executable** |
|---|---|
| `jmain.o` | `jmain.o` |
| `funclist.o` | `funclist.o` |
| | `dbiinit.o` |
| **JAM** Executive Library (JM) | **JAM** Executive Library (JM) |
| **JAM** Screen Manager Library (SM) | **JAM** Screen Manager Library (SM) |
| | **JAM/DB*i*** Common Interface Library (DM) |
| | **JAM/DB*i*** Engine-specific Interface Library (1 or more for each DBMS) |
| | DBMS Client Library (1 or more for each DBMS) |

The **JAM/DB*i*** Common Interface Library includes the generic routines supported by all engines. It is the interface between **JAM** and all the engine-specific processing for accessing a database.

The **JAM/DB*i*** Engine-specific Interface Library is also known as the "support routine." An application must have a support routine for each engine the application uses. The support routine contains all the engine-specific code required by **JAM/DB*i***. The **JAM/DB*i*** Common Interface Library calls an engine's support routine to make the appropriate calls to the DBMS Client Libraries.

The DBMS Client Libraries are supplied by the database vendor. These libraries control all programmed access to a DBMS.

# Chapter 3. JAM/DB*i* *Application Development*

Many of the issues involved in developing a JAM/DB*i* application overlap those involved in developing a JAM application. Here we emphasize issues specific to JAM/DB*i* applications. If necessary, you should see the companion chapter in the JAM Overview for more information on topics like control strings, the Screen Editor, and the Data Dictionary Editor.

## 3.1. CREATING AND EDITING APPLICATION SCREENS

Generally, a developer starts creating a new application by creating screens. The developer may use the JAM/DB*i* authoring executable, `jxdbi`, or the JAM authoring executable, `jxform`. In environments where memory is limited, such as MS-DOS, `jxdbi` may be too large and the developer usually must do all development work with `jxform`. If an application screen will be based on a particular table in the database, the developer may use the JAM/DB*i* utility `tbl2f`. This utility creates a JAM screen with a field for each column in the table. JAM assigns field characteristics based on the column's data type. The utility provides a convenient way to develop a maintenance application for a database table, since the utility also creates the JPL procedures for adding, deleting, and updating rows in the table.

3.1.1.
# Mapping Columns to JAM Variables

JAM/DB*i* provides a simple way of moving data back and forth between JAM and a DBMS. JAM/DB*i* transfers a SQL statement from the application to the DBMS. When the DBMS returns values, JAM/DB*i* transfers those values to JAM variables.

A JAM variable is any of the following:

- a JPL variable created with a `vars` statement,
- a screen variable
- an LDB entry (i.e., a data dictionary entry with a scope of 2 or greater)

JAM/DB*i* provides two ways of mapping a database column to a JAM variable: automatic mapping and aliasing.

## Automatic Mapping

By default, JAM/DB*i* automatically maps a column name in a SELECT statement to a JAM variable with the same name. Suppose the current screen `sales.jam` contains a large scrolling array called `item_no`, and the database table `product` contains a column also called `item_no`. Then,

```
sql SELECT item_no FROM product
```

or,

```
sql SELECT product.item_no FROM product
```

would place the values of column `item_no` in the array `item_no`. Note that a column name always maps to an unqualified field name.

If an application executes

```
sql SELECT * FROM product
```

JAM/DB*i* searches for a JAM variable matching each column in the table `product`. If it finds the variable, it writes the column's values to the variable. If it does not, it ignores the column.

## Aliasing

In some circumstances, automatic mapping is undesirable or even impossible. For example, an application may use one screen to show values from two columns with the same (unqualified) name, or a table may have column names that are not valid JAM variable names. In these cases, developers may specify an alias for one or more database columns using the command DBMS ALIAS.[1]

For example, if a table contained a column named `stock^id`, the application could not use automatic mapping because a caret is not a valid character in JAM variable names. The application must set up an alias for the column. For instance,

```
dbms ALIAS "stock^id" stock_id, "company^name" company
sql SELECT stock^id, company^name, dividend FROM stocks
```

JAM/DB*i* would fetch the values of `stock^id` to the alias `stock_id`. It would fetch the values of `company^name` to the alias `company`. (The quotes are used to help JAM/DB*i* parse the column name.) Since no alias was given for the column `price`, JAM/DB*i* would use automatic mapping for this column.

In a DBMS ALIAS statement, a comma separates one column-variable pair from another.

### 3.1.2. Data Validation

JAM provides extensive character edits and field validation. In JAM/DB*i* applications, developers use these features to help end users enter and retrieve data easily. Rather than replacing database rules, these edits supply an additional layer of software between the end user and the DBMS. While the tables' rules will ensure the integrity of entered data, a developer can simplify the end users' task—for example, by creating item-selection screens listing valid data. In addition to providing a better interface, an application that performs some validation at the frontend is also more efficient because it reduces the number of trips to the server.

## 3.2. ERROR HANDLING

Error handling is an essential component of any database application. In developing a database application, there is often a need for two different approaches to error handling. Developers require low level error messages during the development cycle, while end users usually require high level error messages at runtime.

1. Some engines also support aliasing within a SELECT statement. See the section "Using the Engine's SELECT Syntax" on page 82 for more information.

JAM/DB*i* provides several features to assist the developer with these conflicting needs. For any database error, the application has access to a JAM/DB*i* error code and message and an engine error code and message. With the use of a single statement in an application, the developer may alter the way errors are handled and what messages are displayed. It allows the developer to switch easily between running in development mode and prototype mode, and to see the error message appropriate to the mode. The use of two error handlers is not limited to the development cycle. An application may use one error handler for standard endusers and another for the DBA, for instance.

JAM/DB*i* provides several global variables that hold current error and status information. An application does not need to define its own variables to trap this data. The values are accessible from JPL or C.

## 3.3. ITERATIVE APPLICATION TESTING

Unless your environment has memory constraints, you may use the JAM/DB*i* authoring executable to switch between editing with the Screen and the Data Dictionary Editors, and testing with Application Mode. JAM/DB*i* is turned off in the Screen Editor (draw and test modes) to prevent unintended updates to a database. Without any compilation, you may use Application Mode to test control flow and all JPL procedures in the application. If you are using C hook functions, you must compile and link before testing them.

*Chapter 4.*

# JAM/DB*i* Control Flow

This chapter discusses data flow in JAM/DB*i* applications. To demonstrate the concepts of JAM/DB*i*, it uses a simple example, presenting how the application appears to an end user, and how it appears to a developer. This application is based on the one presented in the JAM Overview. An engine-specific version is supplied in the JAM/DB*i* `samples` directory.

The application consists of three screens. With the first screen, an end user logs on to the database and chooses an area of interest. The next two screens provide access to employee rows stored in three tables. In the application, we use JPL procedures to perform the following:

- Log on and log off a database.
- Query tables, retrieving a single row of values to a JAM screen.
- Query a table, retrieving multiple rows into scrolling arrays.
- Update values in a table.

All the procedures are written in JPL.

This section is not a summary of the product's features. Instead, it uses a fairly simple example to demonstrate control flow in a JAM/DB*i* application. An understanding of the concepts discussed here will help you understand the rest of this document.

Developers interested in creating their own "quick start" application should consider using the utility `tbl2f` to build a small application. `tbl2f` is documented in the *Reference Guide* in this document.

## 4.1.
# SAMPLE APPLICATION – USER'S VIEW

The first screen presented to the end user is `mainscrn.jam`.

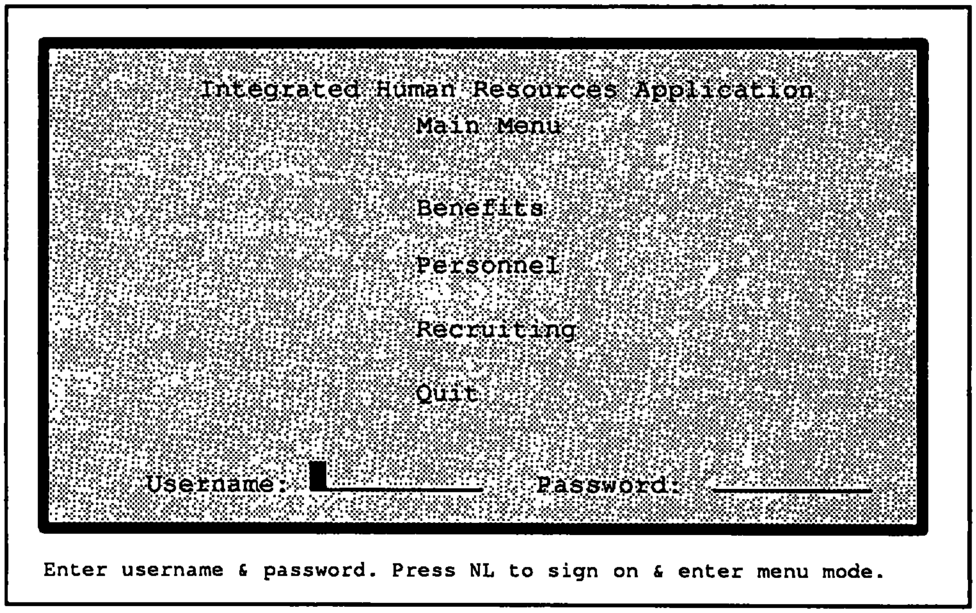



Figure 2: Human Resources Application Main Menu `mainscrn`.

The user must enter a user name and password. If the user has permission to log on, JAM logs on the user and toggles the screen from data entry mode to menu mode. When the user chooses an item from the menu, JAM displays the appropriate screen. If the user chooses `Personnel`, JAM displays the screen `empscrn` shown below.

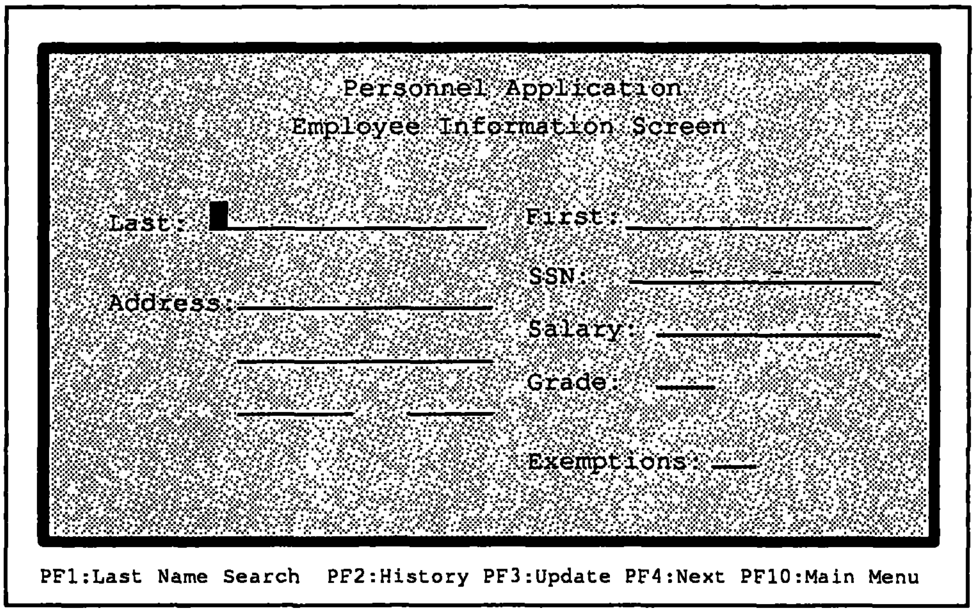



Figure 3: Personnel Application Employee Screen `empscrn`.

The end user enters data in the screen `empscrn.jam` to query the database, and to update rows. The user queries the database by typing an employee surname in the first field and pressing `PF1`. If more than one employee has the same last name, the rows will be retrieved one at a time. The user may press `PF4` to see the next employee row with the specified last name. If the user presses `PF1` without supplying a name, the application retrieves all employee rows in alphabetical order.

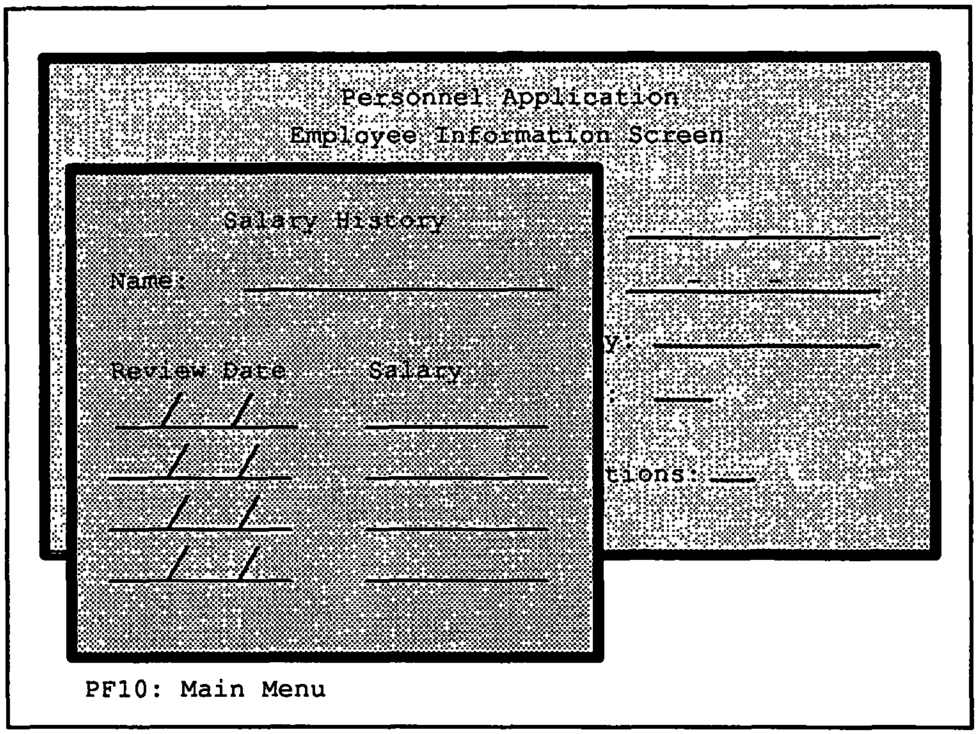



Figure 4: Personnel Application Salary History Window `salhist`.

When **JAM** displays a row, the user may press the `PF2` key to review the employee's salary history.

## 4.2. SAMPLE APPLICATION – DEVELOPER'S VIEW

In this section, we show how the main components of this application appear to a developer. In particular, we describe the database tables, **JAM** screens, and JPL functions constituting the application.

4.2.1.

## Database Tables `emp`, `acc`, and `review`

Below are sample SQL statements for the application tables. Please note that some engines use different names for column datatypes. The table entries represent seven employees.

The table emp has eight columns. Each row stores an employee's social security number, name, home address, and current grade.

```
CREATE TABLE emp (
   ssn       CHAR(11)       NOT NULL,
   last      CHAR(20),
   first     CHAR(12),
   street    CHAR(20),
   city      CHAR(15),
   st        CHAR(2),
   zip       CHAR(5),
   grade     CHAR(1))
```

| ssn | last | first | street | city | st | zip | grade |
|---|---|---|---|---|---|---|---|
| 038–68–6826 | Jones | Barnabus | 321 West 11 St | Albuquerque | NM | 87124 | C |
| 122–98–6541 | Aumond | Hilary | 11–12 Front St | Albuquerque | NM | 87124 | E |
| 122–99–4102 | Jones | Michael | 5 Maple Drive | Albuquerque | NM | 87124 | B |
| 139–42–1651 | Blake | Norman | 34 Concord Ave | Albuquerque | NM | 87124 | D |
| 154–32–6610 | Cory | Richard | 411 Ann St | Albuquerque | NM | 87124 | D |
| 310–77–3997 | Grundy | Janet | 70–2 Poe Ave | Albuquerque | NM | 87124 | D |
| 310–32–0084 | Jones | John P | 9 Vern Terrace | Albuquerque | NM | 87124 | D |

Figure 5: Table emp

Table acc has three columns. Each row stores an employee's social security number, current salary, and a number of tax exemptions.

```
CREATE TABLE acc (
   ssn       CHAR(11)       NOT NULL,
   sal       NUMERIC(10.2),
   exmp      NUMERIC(1))
```

| ssn | sal | exmp |
|---|---|---|
| 038–68–6826 | 29500.00 | 1 |
| 122–98–6541 | 37800.00 | 3 |
| 122–99–4102 | 26000.00 | 3 |
| 139–42–1651 | 89500.00 | 2 |
| 154–32–6610 | 43100.00 | 4 |
| 310–77–3997 | 38000.00 | 1 |
| 310–32–0084 | 47500.00 | 5 |

Figure 6: Table `acc`

Table `review` has four columns. Each row stores an employee's social security number, a hire date or review date, a new salary if it has changed since the previous review, and a new grade if it has changed since the previous review. If `newsal` or `newgrade` is null, the employee was reviewed but there was no change in salary or grade.

```
CREATE TABLE review (
   ssn       CHAR(11)        NOT NULL,
   revdate   DATE            NOT NULL,
   newsal    NUMERIC(10.2),
   newgrade  CHAR(1))
```

| ssn | revdate | newsal | newgrade |
|---|---|---|---|
| 038–68–6826 | 12/13/90 | 49500.00 | C |
| 038–68–6826 | 12/11/89 | 45000.00 | NULL |
| 038–68–6826 | 12/15/88 | NULL | NULL |
| 038–68–6826 | 12/14/87 | 38500.00 | D |
| 122–98–6541 | 04/10/90 | 37800.00 | NULL |
| 122–98–6541 | 04/08/89 | 31000.00 | E |
| 122–99–4102 | 05/01/90 | 29000.00 | NULL |
| 122–99–4102 | 05/01/89 | 25200.00 | E |
| 139–42–1651 | 11/12/90 | 89500.00 | NULL |
| 139–42–1651 | 11/08/89 | 81000.00 | B |
| 139–42–1651 | 11/10/88 | 67500.00 | C |
| 139–42–1651 | 11/10/87 | NULL | NULL |
| 139–42–1651 | 11/08/86 | 53000.00 | D |
| 154–32–6610 | 02/01/91 | 43100.00 | D |
| 310–77–3997 | 07/16/90 | 38000.00 | D |
| 310–77–3997 | 07/14/89 | 30000.00 | E |

| | | | |
|---|---|---|---|
| 310-32-0084 | 03/01/91 | 47500.00 | D |
| 310-32-0084 | 03/01/90 | 43000.00 | E |

Figure 7: Table `review`

The sample application permits an enduser to view rows from these tables and to update data in some columns.

4.2.2.

# Source Module `dbiinit.c`

To save memory, JAM supplies several features as optional subsystems. These subsystems include soft keys and alternate scrolling as well as DB*i*. The JAM/DB*i* subsystem is installed by setting the `DBI` macro in `jmain.c` (or `jxmain.c`) or by setting a compiler directive.

The application must initialize an engine with the function `dm_init` before making a connection. Developers may call this function directly or they may use the vendor structure in `dbiinit.c` to store the engine initialization information. JAM/DB*i* supplies a version of this file customized for your engine.

An excerpt from `dbiinit.c` is shown below. The boldface text shows the statements that would install a fictional DBMS called XYZdb for the sample application.

```
#include "smdefs.h"
#include "dmerror.h"
#include "smusrdbi.h"
#include "dmuproto.h"

#if DBIVENDORLIST
/* Support routine function prototypes */
/* Copy the following line for each support routine */
/* that is to be installed.  Uncomment each copy,    */
/* and replacle 'support_routine' with the name of   */
/* the support routine to be installed.              */

/* extern int support_routine PROTO((int)); */
extern int dm_xyzsup PROTO((int));

/* Add one entry to the following structure for each database support*/
/* routine that is to be installed.  The form of each entry is as    */
/* follows:                                                          */
/*                                                                   */
```

```
/*       { "engine_name", support_routine, case_flag, (char *) 0 },     */
/*                                                                      */
/* Replace 'engine_name' with the name of the database you are         */
/* installing.  Replace 'support_routine' with the name of the         */
/* support routine for that database.  Replace 'case_flag' with        */
/* one of:                                                              */
/*       DM_DEFAULT_CASE         (Use the default value for the        */
/*                                case_flag specified in               */
/*                                the support routine)                 */
/*       DM_PRESERVE_CASE        (No case conversion is performed on   */
/*                                database columns)                    */
/*       DM_FORCE_TO_LOWER_CASE  (Maps upper and mixed case column     */
/*                                names to lower case jam field        */
/*                                names during a database query)       */
/*       DM_FORCE_TO_UPPER_CASE  (Maps lower and mixed case column     */
/*                                names to upper case jam field        */
/*                                names during a database query)       */
/*                                                                      */
/* The last member in the structure is for future expansion.
*/static vendor_t vendor_list[] =
{
/*       { "engine_name", support_routine, case_flag, (char *) 0 },     */
         { "xyzdb", dm_xyzsup, DM_FORCE_TO_LOWER_CASE, (char *) 0 },
         { (char *) 0, (int (*)()) 0, (int) 0, (char *) 0 }
};
```

The entry

```
{ "xyzdb", dm_xyzsup, DM_FORCE_TO_LOWER_CASE, (char *) 0 }
```

contains four elements. The first,

```
"xyzdb"
```

names the engine for the application. It may be any name the developer wishes; an abbreviated vendor name is common. The second element,

```
dm_xyzsup
```

names the engine's support routine. This support routine is supplied in a library as a part of the JAM/DB*i* distribution and its name is documented in the README file. The third,

```
DM_FORCE_TO_LOWER_CASE
```

tells JAM/DB*i* how to handle the case of column names when executing a SELECT. This flag tells JAM/DB*i* to convert column names to lower case when searching for JAM variable destinations for a SELECT. Therefore, the application uses lower case for screen, LDB, and JPL variables that are targets for database columns.

Any developer-written C hook functions are installed in `funclist.c`. Since the sample application uses only JPL it uses the distributed `funclist.c` without any modifications. For more information on `funclist.c` or prototyped functions, see the JAM *Programmer's Guide.*

4.2.3.

## Data Dictionary and Initialization File

The application's data dictionary has three types of entries. They are the following:

- constants named and initialized for JAM/DB*i* errors
- variables for passing database values between screens at runtime

See the figures below.

```
                    DATA DICTIONARY MAINTENANCE

   NAME            SC R/G              COMMENT
 DM_NOCONNECTION_  1  _  Initialized_to_value_of_DBi_code___
 DM_NO_MORE_ROWS_  1  _  Initialized_to_value_of_DBi_code___
 DM_ROLLBACK_____  1  _  Initialized_to_value_of_DBi_code___
 current_ssn______2  _  For_passing_the_value_of_index_key_
 current_name____  2  _  For_passing_the_concatenated__name_
 EOF_____________  _  _  ___________________________________
```

Figure 8: Developer's View of the Data Dictionary.

```
# const.ini
# This file inializes LDB constants.
# Values correspond to those in DBi header file dmerror.h

"DM_NO_MORE_ROWS"      "53256"
"DM_ROLLBACK"          "53263"
"DM_NOCONNECTION"      "53271"
```

Figure 9: Developer's View of the Constants' Initialization File.

The `DM_` variables are named after symbolic constants in the **JAM/DB*i*** file `dmerror.h`. Note that the scope of these variables is 1. At runtime, these values are treated as constants by the local data block (LDB) initialization. A constants' initialization file, such as `const.ini`, assigns the values to the constants. See Appendix B. for the complete list of JAM/DB*i* error codes.

The entries `current_ssn` and `current_name` are used to pass database values between screens at runtime.

4.2.4.

# JAM Screens

There are three application screens.

Each of the screens uses one or more JPL modules. There are several ways of storing and accessing JPL procedures and modules. A module is one or more JPL procedures. The type of module describes how it is stored—in a file, as a miscellaneous field edit, etc. See the JPL Guide in the **JAM** manual for a discussion of these topics.

## Main Screen

The screen `mainscrn.jam` contains a menu and two data entry fields, `uname` and `pword`. The screen opens in data entry mode. The field `pword` has a procedure in its JPL field module. When the end user tabs from this field, the procedure installs an error handler and attempts to log the end user onto the database with the user name and password entered in the fields. If log on is successful, it calls the built-in function `jm_mnutogl` to toggle the screen from data entry mode to menu mode.

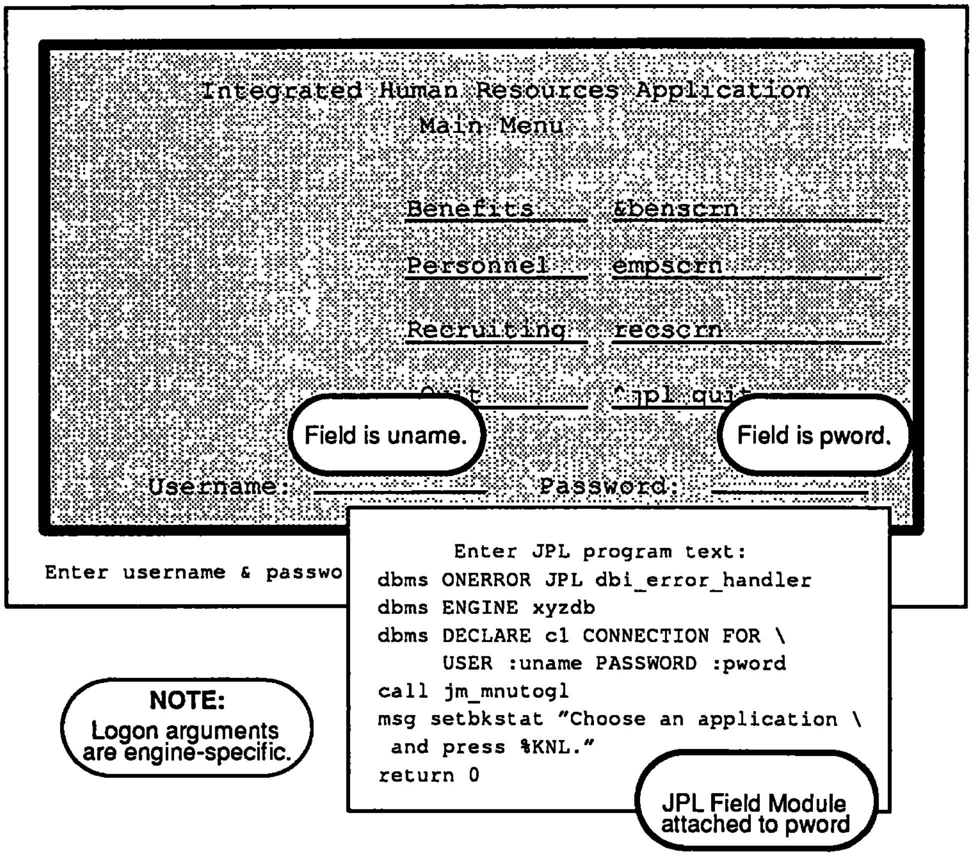



Figure 10: Human Resources Application Main Menu, `mainscrn`.

### JPL Field Module Attached to Field `pword`

The first statement of the procedure sets up error processing for the rest of the application. The DBMS ONERROR statement installs the JPL procedure `dbi_error_handler` as the application's error handler. Whenever a JAM/DB*i* error occurs, JAM/DB*i* passes three arguments to the procedure—the text of the statement that failed, the name of the current engine, and an error flag—and executes the procedure. A sample error handler is shown in Figure 11.

The statement DBMS ENGINE names `xyzdb` as the default engine. Since only one engine was installed in `dbiinit.c`, this statement is optional.

The statement DBMS DECLARE CONNECTION attempts to log the user on to a database server. If log on is successful JPL continues executing the procedure: it toggles `mainscrn` from data entry mode to menu mode and displays a new status line message.

### JPL Procedure for Error Handling, `dbi_error_handler`

If the log on is unsuccessful, **JAM/DB*i*** immediately calls the installed error handler `dbi_error_handler`:

```
proc dbi_error_handler
parms stmt code flag
# All DM_ variables are constants in the LDB.

# If stmt failed because the user did not logon, prompt user to return
# to main screen.

    if (@dmretcode == DM_NOCONNECTION)
    {
        msg emsg "Not logged on. Press %KPF10 to restart."
    }
    else
    {

# For all other errors, display the JAM/DBi message and any database
# error message.

        msg emsg @dmretmsg
        if @dmengerrmsg != ''
            msg emsg @dmengerrmsg
    }

# For all errors, return the abort code (1) to abort the JPL procedure
# where the error occurred. If 0 were returned, the procedure where the
# error occurred would continue executing.
    return 1
```

Figure 11: Sample JPL Error Handler for Human Resources Application.

Note that three arguments are automatically passed to any error handler installed with DBMS ONERROR:

- the text of the statement that failed
- the name of the engine in use when the error occurred
- a flag indicating that this procedure was called because an error occurred

After receiving the arguments, the procedure examines the error code. Note the use of the variables `@dmretcode`, `@dmretmsg`, and `@dmengerrmsg`. These are global variables defined and maintained by JAM/DB*i*. If there is an error executing a `sql` or `dbms` statement, JAM/DB*i* writes a JAM/DB*i* error code to `@dmretcode`, a JAM/DB*i* error message to `@dmretmsg`, an engine-specific error code to `@dmengerrcode` and an engine-specific error message to `@dmengerrmsg`. The application may use these variables in JPL statements such as `if` or `msg` when processing for errors.

The procedure first checks if the user is connected to an engine. For instance, if the user has a mouse and clicks on a menu choice, he or she may move to the next screen before logging on. However, once he or she attempts to view employee data, JAM/DB*i* will return an error because there is no connection to the database. In case of this error, the error handler prompts the user on how to recover—pressing PF10 returns the user to the top-level form where a user name and password may be entered.[2] Recall that `DM_NOCONNECTION` was defined as an LDB constant (Figure 8 and Figure 9).

For all other errors, the error handler displays a standard JAM/DB*i* error message, and also an engine-specific message if there is one in the global JAM variable `@dmengerrmsg`. For example, the user may enter a user name and password on `mainscrn`, but the logon may fail for some reason. In such a case, the handler first displays a JAM/DB*i* message telling the user that the operation failed. Next it displays the engine-specific message further describing the failure—for example invalid user name, password is required, or the server is not available, etc.

In addition to displaying messages, the error handler also determines whether to continue or abort execution of the JPL procedure where the error occurred. If the error handler returns 0, JPL continues execution at the next statement after the one that failed. If the handler returns 1, JPL aborts the procedure and returns control to the procedure's caller.

The sample error handler returns the abort code (1) for all errors. Therefore, if logon fails, JPL does not execute the rest of the procedures in the JPL field module of `pword`. Therefore, it does not execute the statements which toggle the screen to menu mode and change the status line message. Instead, it returns control to the procedure's caller, in this case JAM.

There are many advantages to JAM/DB*i*'s error handling features. Most notably, it gives developers both generic and vendor-specific means of handling errors. In addition, the error handler like the rest of the application is easily prototyped. In early stages of the application, the error handler may simply display all error messages. As the application grows, the developer may enhance the error handler, adding special processing and messages for particular errors. The error handler may also be written in C.

2. Of course, a target list on the menu control strings on `mainscrn` could prevent this. Each menu choice could call a procedure that verifies that the has user logged on before opening the next form or window. See the *Author's Guide* in the JAM documentation for information on using target lists.

To use a JPL error procedure most efficiently, the procedure should be in a public module. See the *JPL Guide* for details.

### Menu Choices on `mainscrn`

Once in menu mode on `mainscrn`, the user may choose among the three applications—`Benefits`, `Personnel`, `Recruiting`—or may quit.

The last option on the menu, `QUIT`, calls the JPL procedure `quit` to log the user off the database and exit the application. Logoff may be executed with the statement,

```
dbms CLOSE_ALL_CONNECTIONS
```

The rest of this chapter describes the `Personnel` option.

## Employee Screen

If the user chooses `Personnel` from the menu, **JAM** opens the form `empscrn` shown below.

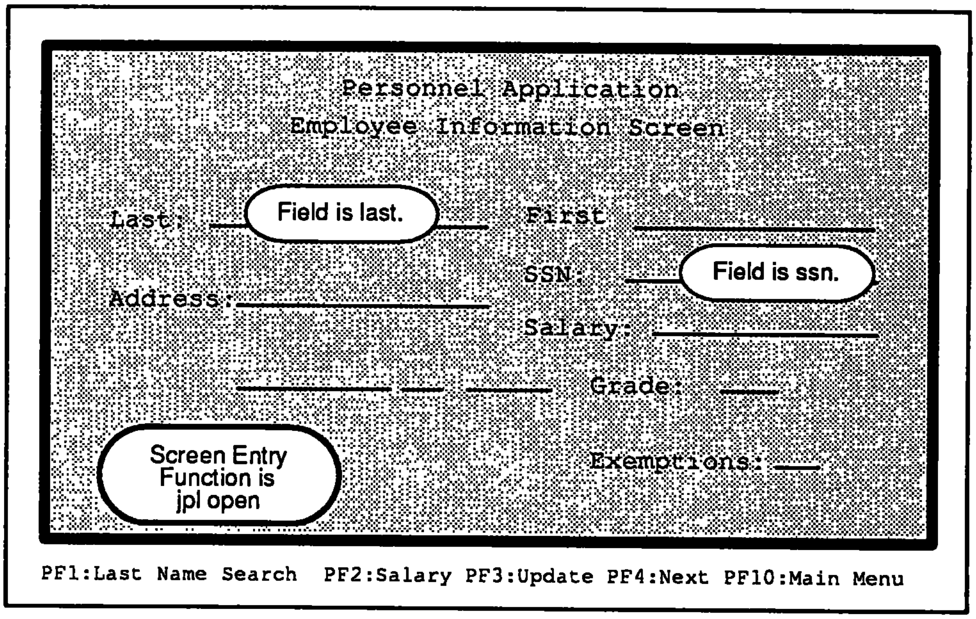



Figure 12: Personnel Application Employee Screen `empscrn`. The social security, salary, and grade fields are protected from data entry.

The screen `empscrn.jam` is used to update and display data from the database. The screen has eleven fields: `last`, `first`, `street`, `city`, `st`, `zip`, `ssn`, `grade`, `sal` and `exmp`. The function keys `PF1` and `PF2` are associated with JPL functions that query the tables `acc`, `emp`, and `review`. The `PF3` key permits a user to update name and address values in the table `emp`, and the number of exemptions in the table `acc`. If the end user wishes to scroll through the employee records, pressing the `PF4` will fetch a new row. The `PF10` key returns the user to the menu screen.

The fields `ssn`, `sal`, and `grade` are protected from data entry. The end user may update an employee's name, address, or number of exemptions. The application assumes that an employee's social security number should not change. An employee's salary and grade may only be changed after an employee review. We assume that such information is entered in another application. Developers, of course, could write a function that permits certain users to change data in protected fields. The **JAM** Programmer's Guide documents the library functions necessary for this type of processing.

Below is the text of the JPL procedures for `empscrn` and an explanation of the procedures.

### JPL Procedure `open`

```
proc open
msg setbkstat "\
%KPF1 Last Name Search %KPF2 Salary  %KPF3 Update  %KPF4 Next \
%KPF10 Main Menu"

dbms DECLARE emp_cursor CURSOR FOR \
SELECT emp.first, emp.last, emp.street, emp.city, emp.st, emp.zip,\
emp.ssn, emp.grade, acc.sal, acc.exmp FROM emp, acc \
  WHERE emp.ssn=acc.ssn AND emp.last LIKE ::parm_last \
  ORDER BY emp.last, emp.first
return 0
```

Figure 13 a : JPL screen module for `empscrn`.

This procedure is the screen entry function. The `msg` statement displays a status line message which describes the screen's control keys. The second statement declares a cursor, `emp_cursor`, for a SELECT statement. The SELECT is just like a SELECT statement executed in a DBMS interface, except for the argument `::parm_last`. This argument is a *binding parameter*. JAM/DB*i* will not know its value until the end user presses the PF1 key which executes the cursor. Executing the cursor will execute the SELECT and fetch data to the screen.

### JPL Procedures `search` and `next`

```
proc search
if last == ""
  dbms WITH CURSOR emp_cursor EXECUTE USING parm_last = '%'
else
  dbms WITH CURSOR emp_cursor EXECUTE USING parm_last = last
if @dmretcode == DM_NO_MORE_ROWS
  msg emsg "There are no employees with the surname :last ."
return 0

proc next
dbms WITH CURSOR emp_cursor CONTINUE
if dbi_retcode == DM_NO_MORE_ROWS
  msg emsg "There are no more rows."
return  0
```

Figure 13 b: Continuation of JPL screen module for `empscrn`. These functions are executed with PF1 and PF4.

The procedure `search` begins by checking if the field `last` is empty. If it is empty, the procedure executes `emp_cursor` (declared in Figure 13 a) using the wild character '%'. Thus, if the end user presses `PF1` without supplying a surname, JAM/DB*i* fetches all the employee rows one at a time in alphabetical order.

If the field `last` is not empty, the procedure executes `emp_cursor` with the surname entered in the field. If two or more employees have the same surname, more than one row is returned. The enduser presses the `Next` key to see the next available record.

For example, if the end user entered the surname "Jones" in the field named `last`, the DBMS would find three qualifying employees in the database. JAM/DB*i* displays the information on employee Barnabus Jones when the `PF1` key is pressed. When the `PF4` key is pressed, JAM/DB*i* displays the next employee in the SELECT set, John P. Jones. When the `PF4` is pressed a second time, JAM/DB*i* displays the information on the final employee, Michael Jones. If the user presses the `PF4` key a third time, the procedure tells the user that there are no more rows in the SELECT set.

The procedure can tell the user when all rows have been displayed because the engine sends a no-more-rows signal if the application tries to fetch more rows than there are in the SELECT set. When this signal is returned, JAM/DB*i* writes the value of the `DM_NO_MORE_ROWS` code to the global variable `@dmretcode`. The JPL procedure knows the value of `DM_NO_MORE_ROWS` because a variable of the same name was defined as an LDB constant (Figure 8) and was assigned a value by the initialization file `const.ini` (Figure 9).

### JPL Procedure `check_ssn`

```
proc check_ssn
 if ssn != ""
   return 0
 msg emsg "\
 A social security number is required. Please enter an employee's\
 last name and press %KPF4 to retrieve the necessary information.\
 When a record is displayed, press %KPF2 to see the salary history\
 or press %KPF3 to make an update."
return 1
```

Figure 13 c: Continuation of JPL screen module for `empscrn`.

The procedure `check_ssn` is used by the procedures `salhist` and `update`. It verifies that the user has entered a social security number. If no number is given, `check_ssn` displays an error message.

### JPL Procedure `salhist`

```
proc salhist
  vars jpl_retcode
  retvar jpl_retcode

  jpl check_ssn
  if jpl_retcode == 0
  {
    cat current_ssn ssn
    cat current_name first " " last
    call jm_keys PF14
  }
  return 0
```

Figure 13 d: Continuation of JPL screen module for `empscrn`.This function is executed with PF2.

The end user presses the `PF2` key to review an employee's salary history. The procedure begins by setting up a return variable and calling the procedure `check_ssn`. The procedure `check_ssn` (Figure 13 c) tests whether the field `ssn` is empty. If `ssn` is empty, the procedure displays a message telling the user to press the `PF1` key before requesting a history. The return code from `check_ssn` determines whether `salhist` continues executing. If the code is 0 (i.e., `ssn` is not empty) the procedure continues.

This routine copies the current employee social security number to the LDB variable `current_ssn`, and concatenates the values of `first` and `last` in the LDB variable `current_name`. The values are copied to the LDB so that the salary history screen may use them.

The statement `call jm_keys` executes a control string. The **JAM** control string window for `empscrn` contains the entry

```
PF14  &(9,25)salhist
```

which opens the screen `salhist` at row 9, column 25. The discussion of the `salhist` screen begins on page 33.

## JPL Procedure `update` and Related Procedures

```
proc update
vars jpl_retcode ans
retvar jpl_retcode
 jpl check_ssn
 if jpl_retcode == 0
 {
    msg query "Update this record now?" ans
    if ans
      jpl tran_handle upd_emp
  }
  return 0

proc tran_handle
parms subroutine
vars tran_error
retvar tran_error
  jpl :subroutine
  if tran_error
  {
     msg emsg "Rolling back transaction."
     dbms ROLLBACK
  }
  else
     msg emsg "Transaction successful."
  return 0

proc upd_emp
  dbms BEGIN
  sql UPDATE emp SET last=:+last, first=:+first, \
    street=:+street, city=:+city, st=:+st, zip=:+zip WHERE ssn=:+ssn
  sql UPDATE acc SET exmp=:+exmp WHERE acc.ssn=:+ssn
  dbms COMMIT
  return 0
```

**NOTE:** Transaction commands are engine-specific.

Figure 13 e: End of JPL screen module for `empscrn`. The procedure `update` is executed with PF4.

The procedure `update` begins by setting up a return variable and calling the procedure `check_ssn`. The procedure `check_ssn` (Figure 13 c) tests whether the field `ssn` is empty. If `ssn` is empty, the procedure displays a message telling the user to press the `PF1` key before performing an update. The return code from `check_ssn` determines whether `update` continues executing. If the code is 0 (i.e., `ssn` is not empty) the procedure continues, asking the user to confirm the update. If the end user enters the value of `SM_YES` (typically "y"), the procedure passes the name of a subroutine upd_emp to a transaction handler `tran_handle`.

The procedure `tran_handle` is a generic procedure that may be used to execute any transaction. It receives one argument, the name of a subroutine that contains the transaction statements. Before calling the subroutine, however, `tran_handle` defines and declares a return variable `tran_error`. After calling the subroutine, `tran_handle` checks if `tran_error` is non-zero; a non-zero value signals that an error has occurred and that `tran_handle` must execute a rollback. This method permits the application to test and rollback for both JAM and JAM/DB*i* errors. The return code for a JAM error is always −1, and the return code from the sample error handler `dbi_error_handler` is 1.

The procedure `upd_emp` is engine-specific. Some engines, such as ORACLE, begin a transaction with the command DBMS AUTOCOMMIT OFF. If you are building this application, please consult the engine-specific documentation.

Note the use of :+***variable*** in the UPDATE statements. This is the colon-plus preprocessor. Before executing the statement, JPL replaces each instance of :+***variable*** with the value of ***variable*** in a format suitable for the engine.

For example, if the screen contained the following values,

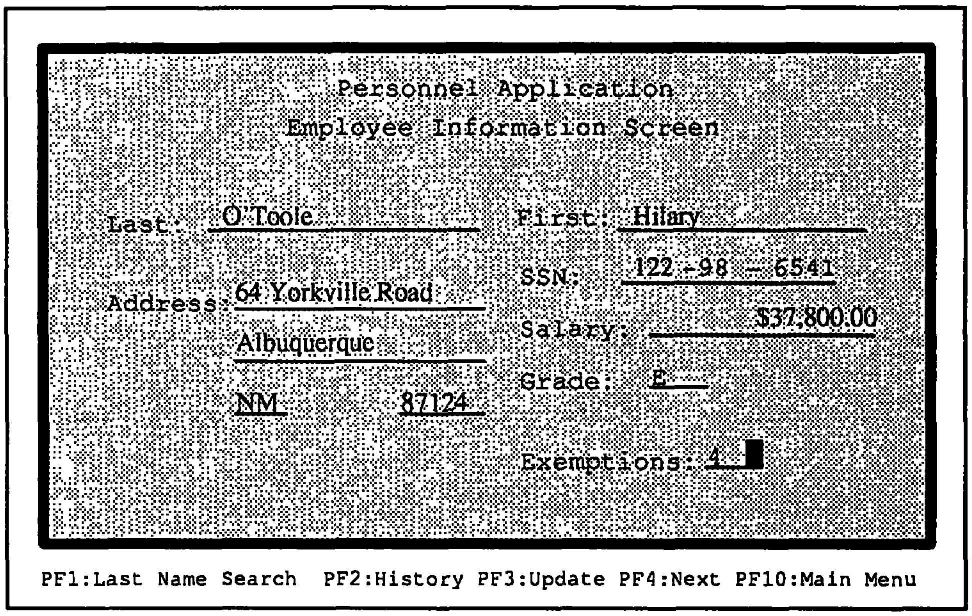



Figure 14: Screen Editor Entry Screen

and assuming that the fields `last`, `first`, `street`, `city`, `st`, and `zip` are all character fields with no special edits, and `exmp` is a `digits only` field, the procedure would execute something like the following,

```
UPDATE emp SET last='O''Toole', first='Hilary', \
   street='64 Yorkville Road', city='Albuquerque',\
   st='NM', zip='87124' WHERE ssn='122-98-6541'

UPDATE acc SET exmp=4 WHERE acc.ssn='122-98-6541'
```

Note that the colon-plus processor formats character data differently than numeric data. Character strings are automatically enclosed in quotes and embedded quotes in character strings are escaped. Numeric values are not quoted. This formatting is engine-specific and is handled automatically by JAM/DB*i*. This topic is covered in detail in the *Developer's Guide* of this manual.

## Salary History Screen

If the user presses the `Salary History` key while an employee row is displayed, JAM opens the window `salhist`, shown below.

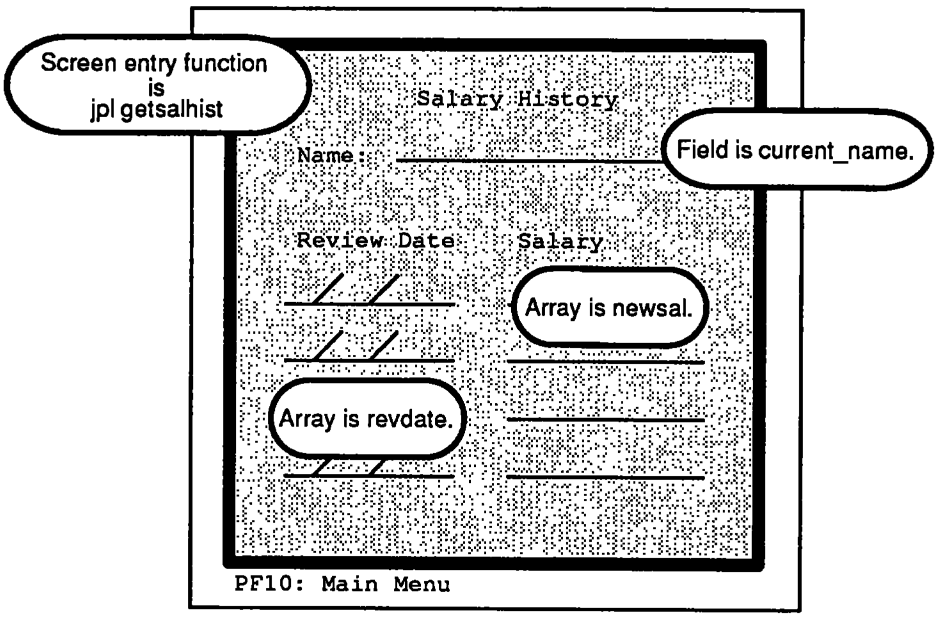



Figure 15: Developer's View of `salhist`.

Upon opening `salhist`, JAM calls the JPL function `getsalhist`, shown below.

```
proc getsalhist
  msg setbkstat "    %KPF10 Main Menu"
  sql SELECT revdate, newsal FROM review WHERE ssn=:+current_ssn
return
```

Figure 16: Developer's View of the JPL Screen Module for `salhist`.

Remember that `current_name` and `current_ssn` are LDB variables (Figure 8). The procedure `salhist` on the previous screen concatenated the values of `first` and `last` in the variable `current_name`, and copied the social security number from `ssn` to `current_ssn` (Figure 13 d). The field `name` is protected from data entry and tabbing.

If `empscrn` is displaying the data belonging to the employee Barnabus Jones when the `History` key is pressed, then `getsalhist` executes

```
SELECT revdate, newsal FROM review \
   WHERE ssn='038-68-6826'
```

and JAM displays the following data:

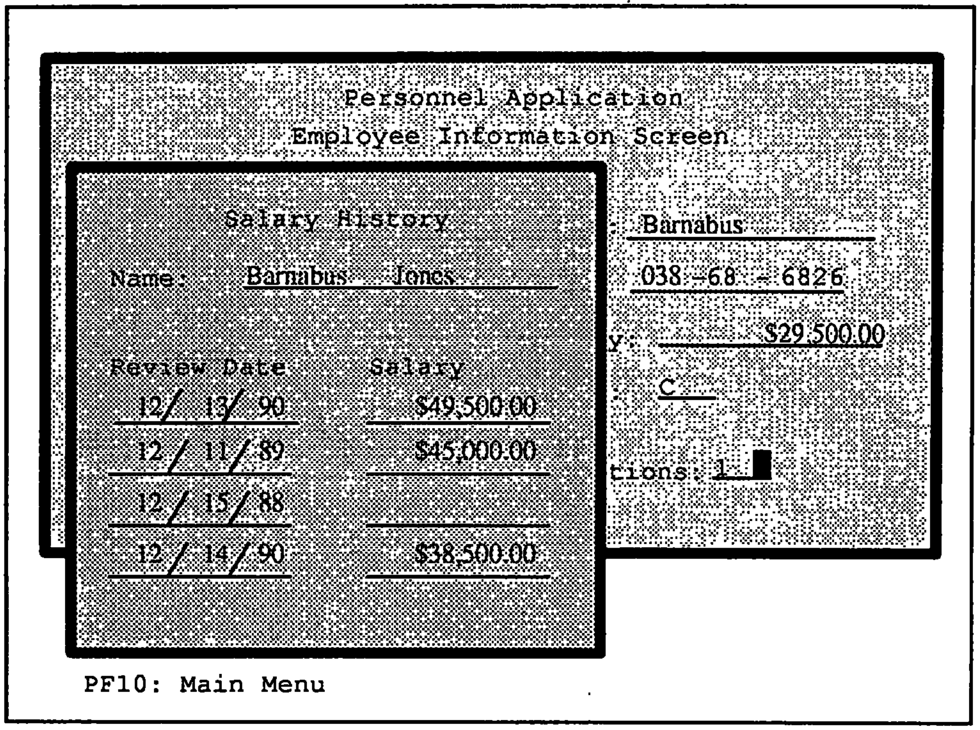



Figure 17: Personnel Application Salary History Window `salhist`.

The arrays `revdate` and `newsal` are large scrolling arrays. The user may press the page-up and page-down keys (JAM logical keys `SPGU` and `SPGD`) to view all the rows. The user may press the `EXIT` key to return to `empscrn`, or press the `Main Menu` key to return to the application's first screen.

## 4.3. JAM/DB*i* CONTROL FLOW SUMMARY

In this section we review control flow between JAM and a database, using the Personnel Application as an example.

In JAM/DB*i* applications, database queries are embedded in hook functions written in JPL or C. Hook functions are explained in detail in the JAM Programmer's Guide. Here we note that the choice of hook function and the choice of coding language affects the construction and the control flow of a query.

4.3.1.
## Variable Substitution

Applications usually require that the end user specify search criteria at runtime. In these cases, an end user enters data into screen fields and JAM uses the fields' contents in the SELECT statement. JAM provides several ways of accessing field contents at runtime. They are the following:

- colon preprocessor
- `sm_getfield` and related functions
- argument of a field function

The colon preprocessor is an easy and efficient method of accessing field contents at runtime. JAM invokes the colon preprocessor on the arguments of a control string beginning with a caret. Therefore, developers may pass the contents of JAM variables as parameters to the control function. If the control string is passing more than one parameter to a C function, the function should be installed as a prototyped function. See the Author's Guide for more information on colon preprocessing and control strings. See the Programmer's Guide for information on prototyped and control string functions.

JAM invokes the colon preprocessor each time it executes a JPL statement. Therefore, JPL developers may access field and LDB values within a JPL procedure. (See the JPL Guide for information on colon preprocessing with JPL commands.)

JAM also invokes the colon preprocessor on the arguments of the JAM/DB*i* library functions `dm_sql` and `dm_dbms`. In addition, C developers may use the library function `sm_getfield`, or a host of variants, to access runtime values. See the Programmer's Guide for descriptions of these JAM functions.

In JAM/DB*i* applications, colon preprocessing is usually preferable to the functions like `sm_getfield` because it automatically formats data in an engine's style.

4.3.2.
## Cursors

SQL vendors support cursors as a part of the interface to custom applications such as `jamdbi`. A cursor is a SQL object that allows an application

- to fetch rows from a SELECT set incrementally
- to use more than one SELECT set at a time

- to improve efficiency when executing a SQL statement many times

On each connection, JAM/DB*i* automatically creates a cursor for SELECT statements. For some engines, it also creates another cursor non-SELECT statements. These cursors are known as the "default" cursors. The JPL command `sql` and the library function `dm_sql` always use a default cursor.

In addition, developers may declare cursors with the command DBMS DECLARE CURSOR. A declared cursor is always named and associated with a SQL statement. Named cursors are executed with the JPL command `dbms` or with the library function `dm_dbms`. In JPL, the statement is

```
dbms WITH CURSOR cursor EXECUTE
```

Executing a named cursor executes the statement that was associated with the cursor at its declaration.

## Fetching a SELECT Set Incrementally

When creating screens for displaying database values, the developer may, at best, only approximate the number of rows which will be in a SELECT set fetched by the application. Therefore, JAM/DB*i* needs a mechanism for handling SELECT sets that contain more rows than can be held by the JAM destination variables at one time. If, for example, a SELECT set contains 100 rows, but destination variables have only twenty occurrences each, JAM/DB*i* cannot fetch more than 20 rows at a time. Therefore, it needs a "place holder" in the set so that after fetching rows 1 through 20 when the SELECT is executed, it can fetch rows 21 through 40 when DBMS CONTINUE is first executed, rows 41 through 60 when DBMS CONTINUE is executed a second time, and so on. A cursor acts as such a placeholder.

## Using Multiple SELECT Sets

JAM/DB*i* automatically creates one default cursor for SELECT statements. Very often, however, applications use two or more SELECT sets concurrently. This would permit a user, for example, to select many item "summary" rows where he or she may position the screen cursor and then execute one or more SELECTs for "detail" rows further describing the item. After viewing detail rows, the user may contain viewing the item summary rows.

This was the approach in the sample application where we used a named cursor to select employee rows and the default cursor to select salary details on an individual employee. This permitted the end user to switch between SELECT statements. If the user pressed the PF1 key without specifying a last name, the application selected all the rows. While scrolling through the rows (pressing the PF4 key), the user was also permitted to view each employee's salary history before viewing the next employee row. If the application did not use a named cursor to select employee rows, JAM/DB*i* would use the default cursor again, losing the user's place in first SELECT set when it issued the second SELECT statement.

## Improving Efficiency

Before executing a SQL statement, the DBMS must prepare the statement. Preparation may include parsing the statement and declaring an engine-cursor. If a statement will be executed many times, declaring a cursor may improve the application's efficiency because the preparation is done only once, rather than each time the statement is executed. An application may declare some cursors upon start up or upon screen entry, and it may use function keys to call procedures which execute the named cursors.

4.3.3.

# Error Processing

JAM/DB*i* provides two ways of managing errors in an application. The default method writes error messages to the status line, just as for JAM errors, and aborts the JPL procedure it was executing. The other method is for the developer to write and install an error handler which JAM/DB*i* will execute whenever a JAM/DB*i* error occurs.

An error handler written in JPL is installed with the statement

`dbms ONERROR JPL` ***procedure_name***

An error handler written in C must be a prototyped function (i.e. installed in `pfuncs` in `funclist.c`) and is installed with the statement

`dbms ONERROR C` ***function***

When a JAM/DB*i* error occurs, JAM/DB*i* will execute the installed error handler. JAM/DB*i* automatically passes arguments to the error handler—the text of the statement that failed, the engine name, and an error flag. The engine name is the name that was used to initialize the engine in `jmain.c`. The error flag equals 2.

The error handler is responsible for displaying any error messages. It may use `@dmretmsg` to display a JAM/DB*i* message, `@dmengerrmsg` to display an engine-specific error message, or it may examine the error codes `@dmretcode` and `@dmengerrcode` and display its own error messages.

The procedure's return code determines whether or not JPL continues or aborts the procedure it was executing.

Error handling is summarized in the figure below.

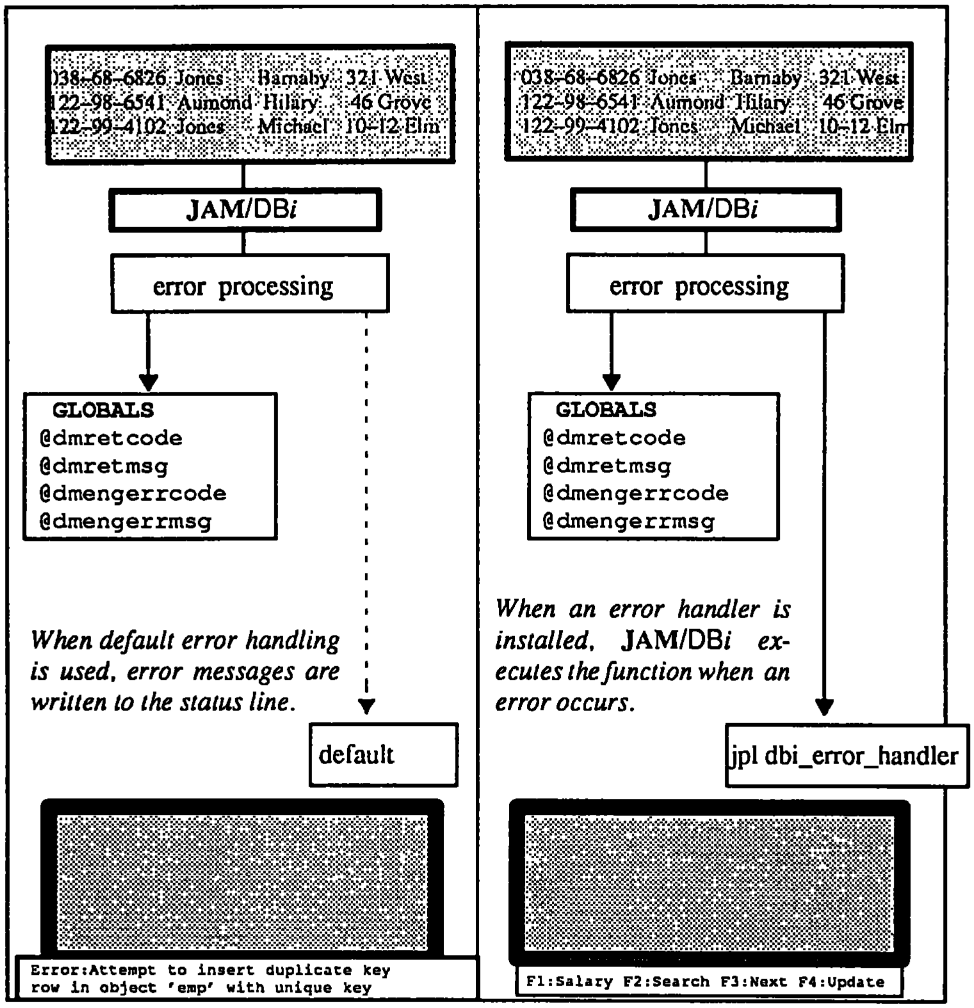



Figure 18: JAM/DBi Error Flow from the Database to JAM. The solid line shows the path used by the example.

# *Chapter 5.* JAM/DB*i* ***Philosophy***

In this chapter, we address several features of JAM/DB*i* and suggest some development strategies.

## 5.1. JAM/DB*i* FEATURES

JAM/DB*i* is a powerful tool for developing frontend applications and interfaces. The sections below discuss its prominent features.

### 5.1.1. SQL–Based

SQL (Structured Query Language) is the standard for relational database languages. It is a tool which provides interactive users with a non-procedural, easy-to-use means of accessing databases and it assumes little or no programming skills. A key feature of JAM/DB*i* is that it uses the SQL syntax of the database you are using. You have complete access to all the features supplied by your DBMS. You do not need to learn a new syntax to use JAM/DB*i* because any SQL statement may be embedded in JPL and C hook functions. In JPL, a SQL statement is prefixed with the verb `sql` or associated with a declared cursor. In C, a SQL statement is passed as an argument to the JAM/DB*i* library function `dm_sql`.

As a result, JAM/DB*i* developers may create an entire frontend application simply using SQL and the JAM authoring tools.

5.1.2.
## OS Portability

JAM/DB*i* is available on most operating system platforms. The JAM terminal and keyboard translation files provide all the hardware configuration needed by JAM/DB*i*. Developers customize the makefile distributed with JAM/DB*i* for software and operating system specifics.

5.1.3.
## Vendor Independence

Vendor independence is an important feature of JAM/DB*i*. Since JAM/DB*i* is available for many popular relational databases, developers may choose a database for its data management capabilities while using JAM's powerful tools to create the frontend applications. In this way, developers are not limited by the vendor's frontend development tools.

In addition, JAM/DB*i* provides a standard means of moving applications from one database to another, with no changes to screens. If the two databases use different SQL syntax, however, developers may need to make some changes to SQL statements. Additional changes may be needed for differences in locking and transaction management on the two databases.

5.1.4.
## Multi-engine Support

Some installations may maintain several databases, each with a DBMS supplied by a different vendor. JAM/DB*i* permits developers to access different engines in the same application. The user must have a JAM/DB*i* support routine for each DBMS product that the application will use.

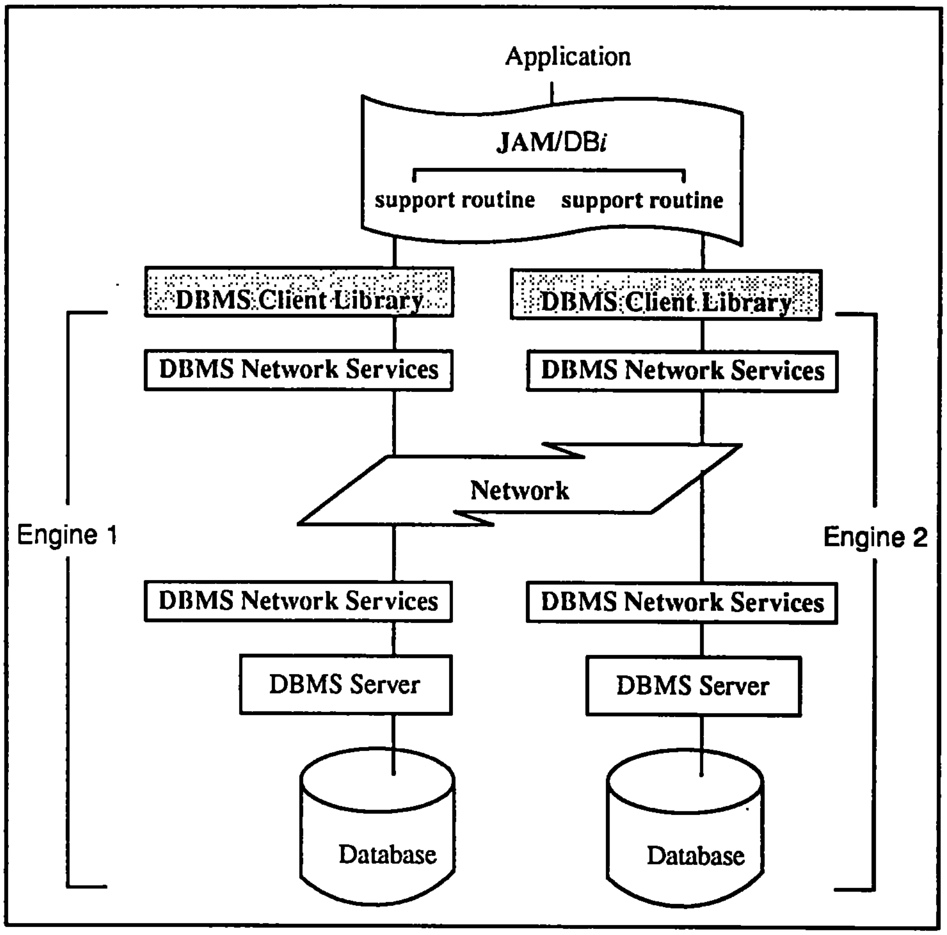



Figure 19: Components of JAM/DB*i* Architecture when using multiple engines.

5.1.5.

## Multi–connection Support

Some engines permit multiple connections. This allows an application to have connections to multiple servers and databases of the engine. Connections are named, permitting the application to set a default connection and to switch between connections as it executes database operations.

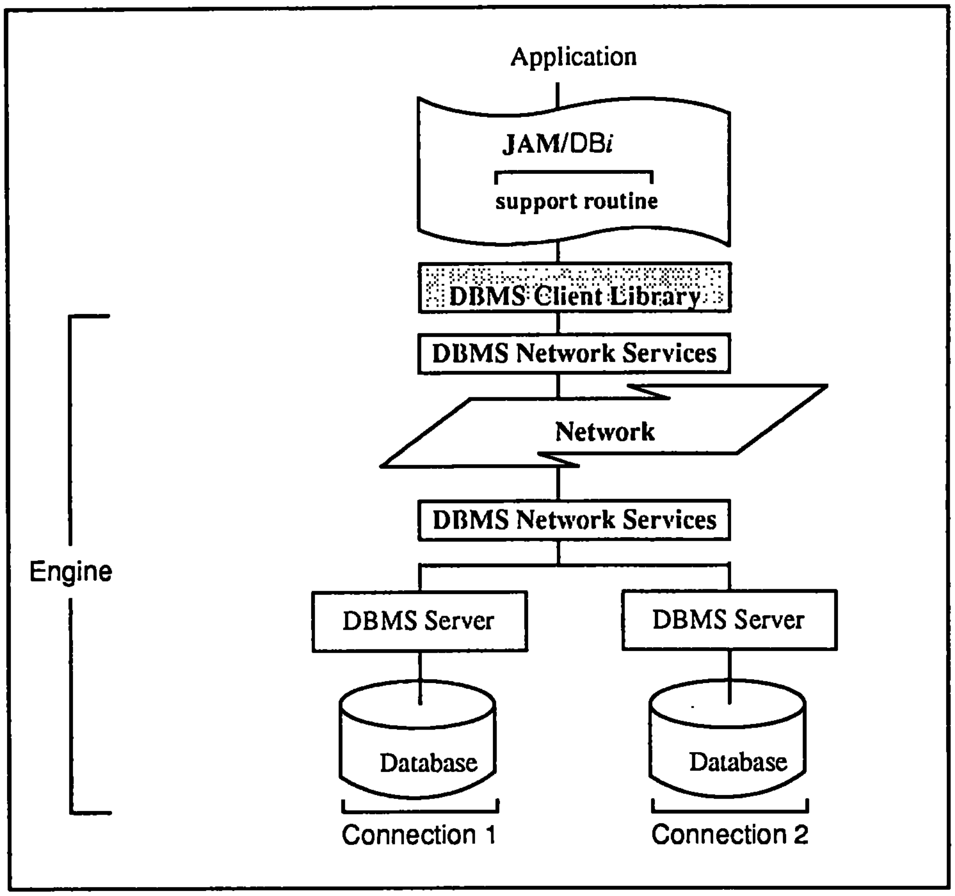



Figure 20: Components of JAM/DB*i* Architecture when using multiple connections.

5.1.6.

# Prototyping

Developers using JAM/DB*i* may prototype an application with real links to a database without writing any third-generation programming code. Database functions may be simulated by placing sample data on screens with JPL. Later, the the simulation code can be replaced with `sql` and `dbms` statements.

## 5.2. JAM/DB*i* DEVELOPMENT HINTS

There are a few suggestions which developers should consider before developing an application.

- Execute SELECT statements when the target variables are on the active screen. Use the LDB just to pass a particular column value to another screen when necessary. In the sample application, two screens needed the values of the employee's social security number, first name, and last name. Rather than putting the target variable `ssn` in the data dictionary, the application defined `ssn` on the screen `empscrn` and defined `current_ssn` in the data dictionary. Therefore, `current_ssn` contains a value only when the application explicitly writes to the variable. By keeping only necessary column variables in the LDB, the developer reduces the amount of memory needed by the LDB, reduces the chances that the LDB will pass data to an unexpected target, and reduces the amount of application maintenance.
- Use an error and/or exit handler to process error and status information. Not only does this reduce the amount of code in the application, it also ensures consistent error handling throughout the application.

Appendix C. covers these topics in more detail.

# Developer's Guide

# Chapter 6. Introduction to Development

This document is intended for JAM/DB*i* developers. We discuss the development and creation of executable JAM/DB*i* programs using developer-written hook functions to access and manipulate a database.

We assume that the reader is familiar with JAM. JAM/DB*i* developers should see the JAM *Author's Guide* for information on using the Screen Editor, Keyset Editor, and the Data Dictionary Editor. They should see the *JPL Guide* for information on writing and storing JPL procedures. They should see the *Programmer's Guide* for information on installing C hook functions in the application function list and for customizing the source modules, `jmain.c` or `jxmain.c`.

In addition, developers should review the JAM *Development Overview* and the JAM/DB*i* *Development Overview* before proceeding. These sections discuss the architectural components and the control flow of JAM and JAM/DB*i*.

## 6.1. SQL VARIANTS

SQL is an evolving standard in the database industry and there are numerous SQL-based products on the market today. At this writing JAM/DB*i* supports more than ten vendors' SQL-based products. Each of these vendors implements aspects of SQL differently. For example, some engines permit the use of only single quotes around literals in query statements. Other engines permit the use of either single or double quotes. Engines often have different rules for the use of case and special characters in variable names. JAM/DB*i* provides features to assist developers with these differences. Developers may use the colon-plus preprocessor to format values for a particular DBMS engine before inserting them in database columns. They may control case handling by setting the engine's case flag at initialization.

The obvious advantage is ease of use. JAM/DB*i* provides access to almost all functions supported by the vendor, without changes in command syntax. Developers concerned with DBMS portability, however, must use a compatible SQL syntax. For example, the SQL syntax of most vendors includes a subset of ANSI-compliant SQL commands. The syntax of these commands is usually portable.

The *Developer's Guide* discusses concepts common to all supported engines. For this reason, we do not emphasize any particular implementation of SQL. Any SELECT, INSERT, UPDATE, or DELETE statement in the examples is used only to clarify concepts. When using the concept in an actual application, use the SQL syntax of the DBMS.

## 6.2. JAM/DB*i* COMMANDS

Developers may execute JAM/DB*i* functions from JPL statements and C language function calls. JAM/DB*i* distinguishes between two types of database commands. In JPL, database commands are executed with either the command `sql` or the command `dbms`. Similarly in C, database commands are executed with the functions `dm_sql` or `dm_dbms`.

The `sql` variants execute statements that may be given in the interactive query language of the database. They include CREATE, DROP, SELECT, INSERT, UPDATE and DELETE.

The `dbms` variants execute the following types of functions:

- Statements not needed or not supported in the database's interactive query language. (i.e., LOGON, DECLARE CURSOR, CONTINUE)
- Statements to customize the JAM/DB*i* environment. These include error trapping and directing output to a file or an array occurrence.
- Vendors' "extended" SQL functions. These functions are non-standard enhancements to SQL (e.g., browse, control execution of a stored procedure, etc.).
- SQL statements to be executed under the control of explicitly declared cursors.

Actually, any SQL statement may be executed with a `dbms` command. This is done in two steps: a cursor is declared and associated with the SQL statement, and then the cursor is executed. Developers may use the "short-cut" command `sql` to execute simple queries in a single step. For example,

```
dbms DECLARE item_cursor CURSOR FOR \
    SELECT description, price FROM products \
    WHERE code = :+code
dbms WITH CURSOR item_cursor EXECUTE
```

fetches the same rows as

```
sql SELECT description, price FROM products \
   WHERE code = :+code
```

6.2.1.
## JPL versus C

The colon preprocessor has always been a powerful incentive to use JPL rather than C for JAM/DB*i* functions. Release 5 makes two improvements to the colon preprocessor: it provides a special form for formatting database values, and it performs colon preprocessing on the arguments of `dm_dbms` and `dm_sql`, the library functions for executing database commands.

The decision to use JPL or C is left to the developers' discretion. Developers should know that they may execute any SQL statement from either language, and they may use either or both languages in an application. JPL procedures may be executed without compilation.

Most of the examples in this guide use JPL.

# Chapter 7.
# Access and Execution

In this chapter we discuss how an application accesses and queries a database. We discuss the following topics:

- Initializing one or more engines – the application tells JAM/DB*i* which engines (i.e., vendor products) it will use. (Section 7.1.)
- Connecting to a server and database – the application connects to a server where an initialized engine is running. (Section 7.2.)
- Using cursors – the application uses a default or named cursor to execute an operation on a connection. (Section 7.3.)

# 7.1. INITIALIZING ONE OR MORE ENGINES

An *engine* is a DBMS product. It is identified by a specific vendor and version. For example, SYBASE 4.0, ORACLE 6.0, and ORACLE 5.1 are three distinct engines. JAM/DB*i* is distributed with an object file containing a support routine for a particular engine. The support contains all the vendor-specific code necessary for executing database operations with JAM/DB*i*.

JAM/DB*i* permits an application to access one or more engines. The application must have a support routine for each engine, and it must initialize an engine before opening a connection or a executing a query on the engine.

## 7.1.1. Initializing an Engine in `dbiinit.c`

A call to initialize one or more engines may be put in the JAM/DB*i* source module `dbiinit.c`. A sample `dbiinit.c` is distributed with JAM/DB*i*. The file,

1. makes a function declaration for one or more support routines
2. describes the engine initialization in the structure `vendor_list`

`vendor_list` appears like the following,

```
static vendor_t vendor_list[] =
{
    {"engine", support_routine, case_flag | error_flag, (char *) 0},
    {(char *)0, (int (*)()) 0, (int) 0, (char *) 0}
};
```

The name for ***engine*** is chosen by the developer. If an application uses two or more engines, the application will use the mnemonic ***engine*** to tell JAM which DBMS to use. Most of the examples in the guide use a vendor name as the mnemonic, for example `sybase` or `oracle`, but any character string that is not a keyword is valid. Keywords are listed in Appendix A.

The name of ***support_routine*** is documented in the distributed `dbiinit.c`. The name is usually in the form dm_***vendor***sup where ***vendor*** is an abbreviated vendor name. Some examples are

- `dm_intsup`
- `dm_orasup`

- dm_sybsup

***case_flag*** sets the case-handling feature of JAM/DB*i*. It determines how JAM/DB*i* uses case to map column names to JAM variables when executing a SELECT. The options are

| | |
|---|---|
| ■ DM_PRESERVE_CASE | Use case exactly as returned by the engine. |
| ■ DM_FORCE_TO_UPPER_CASE | Force all column names returned by an engine to upper case. The developer should use upper case when naming JAM variables. |
| ■ DM_FORCE_TO_LOWER_CASE | Force all column names returned by an engine to lower case. The developer should use lower case when naming JAM variables. |
| ■ DM_DEFAULT_CASE | Usually defaults to DM_PRESERVE_CASE. Another default value may be set by JYACC in the support routine. |

For example, ORACLE returns all column names in upper case. If DM_PRESERVE_CASE is set, JAM/DB*i* will look for JAM variables with upper case names. To map columns to JAM variables with lower case names, set the case flag to DM_FORCE_TO_LOWER_CASE. SYBASE, on the other hand, is case sensitive and it may return column names in upper, lower, or mixed cases. To map SYBASE columns to single case JAM variables, set the case flag to DM_FORCE_TO_UPPER_CASE or DM_FORCE_TO_LOWER_CASE.

***error_flag*** determines which error messages are displayed by the default error handler. This flag is "or–ed" with the case flag. The options are

| | |
|---|---|
| ■ DM_DEFAULT_DBI_MSG | The default error handler displays engine-independent error messages when an error occurs. These messages are defined in the JAM message file. |
| ■ DM_DEFAULT_ENG_MSG | The default error handler displays engine-dependent error messages when an error occurs. These messages are supplied by the engine. |

If neither flag is used, the default is DM_DEFAULT_DBI_MSG.

The last argument (char *) 0 is provided for future use.

If the `DBI` subsystem is installed (i.e., its macro is set to 1 in `jmain.c` or by a compiler directive), `jmain` (or `jxmain`) will call the JAM/DB*i* library function `dm_init` for each support routine in the list.

If the initialization is successful, ***support_routine*** returns zero. In some cases ***support_routine*** may reject the initialization and return an error code. In these cases, there may be insufficient memory, the engine may not be installed, or the application may have initialized the same support routine more than once. If such an error occurs when executing `jmain`, JAM will display an error message and terminate.

7.1.2.
## Initialization Procedure

As a part of initialization, JAM/DB*i* calls the support routine for information on the particular DBMS. For each ***engine***, JAM/DB*i* has information on the following

- the engine's capabilities (e.g., whether the engine can execute stored procedures or support multiple connections)
- the required formatting for character and null strings being inserted into a table
- the default for case handling

In addition, JAM/DB*i* sets up some structures at initialization, including structures for tracking the number and names of all connections on an engine.

7.1.3.
## Setting the Default Engine

The application may connect to any initialized engine.

An application with two or more initialized engines sets the *default engine* with the command

`DBMS ENGINE` ***engine***

or sets a *current engine* for a statement with the clause `WITH ENGINE`. An application accessing multiple engines must reset the default or current engine when declaring connections to the different engines. Once a connection is declared, the default connection determines the default engine.

# 7.2. CONNECTING TO A DATABASE SERVER

Before performing operations on database tables, JAM/DB*i* must connect to a DBMS server with the statement

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
    FOR OPTION argument [OPTION argument]
```

Different engines support different options. Please see the DBMS-specific *Notes* in this document for a list of the valid options.

Once a connection is opened, the application may operate on the database tables.

A declared connection is a named structure describing a session on an engine. This information includes

- a connection name
- a pointer to engine information
- logon information supplied by the option arguments, for example, a user and database name
- a data structure for a default SELECT cursor
- pointers to other structures associated with the connection, including named cursors (thus when an application closes a connection, JAM/DB*i* is able to close all open cursors on the connection)

If no engine is named, the connection is declared for the default engine.

The statement

```
dbms CLOSE CONNECTION connection
```

logs off and closes the connection.

## 7.2.1. Connections to Multiple Engines

If an application is using two or more engines, a connection may be declared for each engine. A default connection may be set with the command

```
dbms CONNECTION connection
```

For example,

```
dbms WITH ENGINE sybase DECLARE sybcon CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER birch
dbms WITH ENGINE oracle DECLARE oracon CONNECTION FOR \
   USER :uname PASSWORD :pword
dbms CONNECTION sybcon
sql SELECT * FROM emp WHERE last = :+last
```

In the example, connections are declared on the engine `sybase` and the engine `oracle`. The connection `sybcon` is chosen as the default. Therefore, JAM/DB*i* performs the SELECT on the connection `sybcon` and uses the support routine of `sybcon`'s engine to execute the query.

The WITH CONNECTION clause specifies a connection to be used for a single statement, overriding the default connection. For example,

```
sql WITH CONNECTION oracon SELECT * FROM sales
```

Remember that a connection is always associated with an installed engine. Setting a connection as the current or default connection also sets the current or default engine.

7.2.2.

## Multiple Connections to a Single Engine

Some engines permit two or more simultaneous connections. See the DBMS-specific *Notes* in this document for information on your engine. Developers who wish to take advantage of this feature on a valid engines should declare a named connection for each session on the engine.

```
dbms ENGINE sybase
dbms DECLARE s1 CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER birch
dbms DECLARE s2 CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER maple
dbms CONNECTION s1
```

If this is the second or later connection on the engine, and the engine supports multiple connections, the support routine opens the additional connection and JAM/DB*i* keeps a count of the number of active connections for the engine. If the engine does not support multiple connections or the connection name is not unique, JAM/DB*i* returns the error `DM_ALREADY_ON`.

The application may close all connections by executing DBMS CLOSE CONNECTION for each declared connection or it may close all connections on an engine or all engines by executing

```
dbms [WITH ENGINE engine] CLOSE_ALL_CONNECTIONS
```

# 7.3. USING CURSORS

A *cursor* is a SQL object associated with a specific query or operation. JAM/DB*i* stores information on each cursor. This includes,

- the cursor's name
- the cursor's connection
- any cursor attributes assigned with the commands `DBMS ALIAS`, `DBMS CATQUERY`, `DBMS FORMAT`, `DBMS OCCUR`, `DBMS START`, `DBMS STORE`, and `DBMS UNIQUE`
- other operation–specific information (e.g., the number of rows to fetch, information on target variables or binding parameters, etc.)

Cursors are not JAM variables, and they do not follow the scoping rules of JAM variables. When a cursor is declared, JAM/DB*i* creates a structure for it and adds its name to a list of open cursors. The cursor is available throughout the application until the application closes the cursor or closes the cursor's connection. JAM/DB*i* frees the structure when the cursor is closed.

Every connection has one or two default cursors which JAM/DB*i* automatically creates. An application may also declare named cursors on a connection. A JAM/DB*i* application may use either or both of these types of cursors.

The default cursors are convenient for SQL statements that are executed once, and for applications using only one SELECT set at a time. All database commands executed with the JPL command `sql` or the library function `dm_sql` use default cursors.

Named cursors are convenient for SQL statements that are executed several times. A cursor is declared for a statement; executing the cursor executes the statement. Named cursors often improve an application's efficiency because the same statement does not need parsing each time it is executed. Named cursors are also necessary for applications using more than one SELECT set at a time.

The rest of this section describes the use of cursors in an application. Please note that the discussion of how data is passed between an application and a database is not covered here but in Chapters 8. and 9.

## 7.3.1. Using the Default Cursor

For most engines, JAM/DB*i* automatically declares two default cursors—one for SELECT statements and one for non-SELECT statements such as UPDATE. In a few cases, the engine's standard is a single default cursor and JAM/DB*i* will declare one default cursor. On such engines, an additional option, CURSORS, is supported in the engine's DECLARE connection statement. It permits the developer to choose between one or two default cursors for the connection. See the DBMS-specific *Notes* in this document for more information.

A default SELECT cursor is associated with a particular connection, namely the connection in effect when a SELECT statement is executed. For example,

```
dbms CONNECTION c2
dbms WITH CONNECTION c1 \
   SELECT code, region FROM sales WHERE sales > 999.99
sql UPDATE sales SET code = :+code WHERE region = :+new
```

The first statement sets the default connection. The second statement uses WITH CONNECTION to set c1 as the current connection for the SELECT statement. In the last statement, no connection is specified for the UPDATE statement. Therefore, JAM/DB*i* uses the default connection c2.

7.3.2.

## Using a Named Cursor

A developer may create one or more named cursors to access and manipulate data. The sequence is the following:

- Declare one or more named cursors.
- Execute cursor(s).
- Close cursor(s).

### Declaring a Cursor

Named cursors are created with a declaration statement. The statement names the cursor and associates it with a connection and a SQL statement. If a connection is not named in the declaration, JAM/DB*i* uses the default connection.

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQLstmt
```

For example,

```
dbms DECLARE customer_cur CURSOR FOR \
   SELECT * FROM directory WHERE lname = :+lname
```

This statement is a declaration statement. JAM/DB*i* does not pass the query to the DBMS. Instead it parses the query, performing any specified colon expansion. Colon expansion is not repeated when the cursor is executed.

# Executing a Cursor

Once a cursor has been created, the statement

```
dbms WITH CURSOR cursor_name EXECUTE
```

executes the SQL statement associated with ***cursor_name***. For the examples used above, the statement

```
dbms WITH CURSOR customer_cur EXECUTE
```

executes the SQL statement `SELECT * FROM directory WHERE lname =` ***value of lname when cursor was declared.*** If qualifying rows are found, the database will return them now to JAM/DB*i*.

If the SQL statement is a SELECT statement that retrieves more rows than will fit on the screen, the statement

```
dbms WITH CURSOR cursor_name CONTINUE
```

continues the previous EXECUTE for ***cursor_name*** by fetching the next screenful of records from the SELECT set.

## Executing a Cursor with Parameters

Parameters may be passed with the statement DBMS EXECUTE. The syntax is the following:

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQL statement
dbms [WITH] CURSOR cursor EXECUTE USING var1 [, var2...]
```

There is a one-to-one mapping between parameters in ***SQL statement*** and the ***var*** values in the USING statement. In a DECLARE CURSOR statement for any engine, JAM/DB*i* interprets `::`***parameter*** as a binding parameter. For example,

```
dbms WITH CONNECTION c1 DECLARE x_cursor CURSOR \
   FOR SELECT * FROM sales WHERE cost = ::parm

dbms WITH CURSOR x_cursor EXECUTE USING newcost
```

Note that the use of parameters is different than the use of colon preprocessing when declaring a cursor. When the colon preprocessor is used, column values are supplied when the cursor is declared. To use different values, the cursor must be redeclared before it is executed. When binding is used, the application supplies column values each time it executes the cursor.

If an engine uses another syntax for binding parameters, JAM/DB*i* will also support it.

This topic is covered in detail in Section 8.2.

### Note to Developers Using Multiple Connections

Note that the command DBMS EXECUTE does not permit the WITH CONNECTION clause. The cursor remains associated with the connection specified by name or by default in the DECLARE statement. For example,

```
dbms CONNECTION sybcon
dbms DECLARE cur1 CURSOR FOR SELECT * FROM books
dbms CONNECTION oracon
dbms WITH CURSOR cur1 EXECUTE
sql UPDATE ....
```

When cursor `cur1` is declared **JAM/DB*i*** associates it with the default connection `sybcon1`. Although the default connection is changed to `oracon` before the cursor is executed, the connection associated with `cur1` does not change. When the cursor is executed, the **JAM/DB*i*** performs the SELECT on connection `sybcon`. The default connection `oracon` performs the subsequent UPDATE.

## Modifying or Closing a Cursor

A cursor may be redeclared for another SQL statement. For example,

```
DBMS DECLARE abc CURSOR FOR \
   SELECT order_id, total FROM newsales \
   WHERE total > :+cost
DBMS WITH CURSOR abc EXECUTE

DBMS DECLARE abc CURSOR FOR \
   SELECT * FROM directory WHERE dept = 'Sales'
DBMS WITH CURSOR abc EXECUTE
```

**JAM/DB*i*** provides several commands for changing the default behavior for a cursor associated with a SELECT statement. The commands are DBMS ALIAS, DBMS CATQUERY with DBMS FORMAT, DBMS OCCUR, and DBMS START. They are discussed in Chapter 9. Here we note that these settings are not lost when a cursor is redeclared, but only when the cursor is closed.

To close a cursor and free its data structure, execute the following

dbms CLOSE CURSOR ***cursor_name***

# Chapter 8.
# Data Flow from JAM

This chapter discusses how JAM/DB*i* passes data from an application to a database. The topics are the following:

- Colon preprocessing: using the colon preprocessor to put JAM values into SQL statements. Its forms are :***variable*** and :+***variable***.
- Parameters: binding values to SQL parameters when executing a named cursor. Their form is ::***variable***.

## 8.1. COLON PREPROCESSING

JAM supports two types of colon preprocessing,

- :var Standard colon preprocessing, and
- :*var Re-expanded colon preprocessing.

Both methods are described in the *JPL Guide* in Volume II of JAM. One or more colon variables may appear almost anywhere in a `sql` or `dbms` statement. There are two exceptions.

The first word in the statement may not be colon-expanded. Therefore, the statements

```
:verb SELECT * FROM students
:command EXECUTE cursor1
```

are both illegal. JPL must know the command word to perform syntax checking and compilation before executing a JPL statement.

Colon expansion is not permitted in the WITH ENGINE or the WITH CONNECTION clause. Therefore,

```
dbms :eng_str DECLARE c1 CONNECTION FOR USER :uname
sql WITH CONNECTION :cname SELECT * FROM students
```

are also both illegal. JPL must know which engine or connection is in use before performing any colon processing.

In addition to the standard forms, JAM/DB*i* supports special forms of colon pre-processing for values sent to a database. The forms are

- :+var Database colon preprocessing for column values (colon-plus)
- :=var Database colon preprocessing for operator and column values (colon-equal)

These forms of colon preprocessing replace a variable with its value and format it in a style that is appropriate for a column value in an INSERT statement, an UPDATE statement, or a WHERE clause. They are described below.

### 8.1.1. Colon-plus Processing

Before colon preprocessing a statement, JPL determines which engine to use. If executing a `sql` or `dbms` statement, the JPL parser examines the statement for a WITH ENGINE clause.

If it finds the clause, it uses the specified engine. If it finds a WITH CONNECTION clause, it uses the connection's engine. If neither clause is used, JPL uses the engine of the default connection. In other JPL statements, such as `cat`, JPL always uses the engine of the default connection. Note that colon-plus processing is not necessary in statements using the WITH CURSOR clause. The only WITH CURSOR statement that uses column values is DBMS EXECUTE and this statement uses binding, not colon-plus processing, to supply column values.

For each :+***variable*** used in the JPL statement, the following steps are performed:

1. The standard colon preprocessor replaces the variable :+***variable*** with the value of ***variable***.
2. The colon-plus processor examines the source. If ***variable*** has a null edit and its value is the null edit's string, the colon-plus processor replaces the value with the engine's null value. If it does not have a null edit, or does not contain the null edit string, the processor determines the variable's JAM *type*. The term JAM *type* refers to a classification of JAM field characteristics used by the library function `sm_ftype`, the colon-plus processor, and JAM/DB*i* routines for binding. The JAM types are
   - `DT_CURRENCY`
   - `DT_DATETIME`
   - `DT_YESNO`
   - `FT_CHAR`
   - `FT_DOUBLE`
   - `FT_FLOAT`
   - `FT_INT`
   - `FT_LONG`
   - `FT_PACKED`
   - `FT_SHORT`
   - `FT_UNSIGNED`
   - `FT_VARCHAR`
   - `FT_ZONED`
3. If the JAM type is `DT_DATETIME`, `FT_CHAR`, or `FT_VARCHAR` the processor formats the value according to engine-specific rules, usually enclosing the string in quote characters. For the other format types, the processor calls a function to strip amount editing characters, such as dollar signs, from the value. Finally, the new value is returned to the JPL statement.

The steps are described in full below.

## Step 1. Perform Standard Colon Preprocessing

JAM will search for ***variable*** in the following places

- JPL variables local to the procedure that JPL is executing
- JPL variables local to the module containing the procedure that JPL is executing
- fields on the current screen
- LDB variables[3]

When it finds the variable, it copies its value to an internal work buffer. Any formatting is performed on this copy. The variable's contents remained unchanged.

For more information on variables and scope, see the *JPL Guide*.

## Step 2. Determine the Variable's JAM Type

If the variable is a field or LDB entry that has a null edit, and the value of the variable equals this null edit string, the processor replaces the value with the engine's null string. On most engines, it is the string `NULL`. For example, if field named `address` had a null field edit, the Screen Editor window could appear as the following:

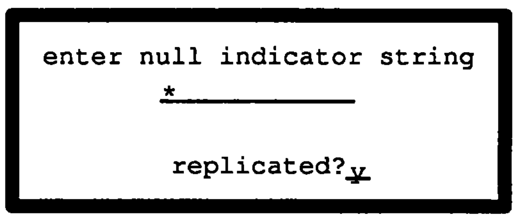


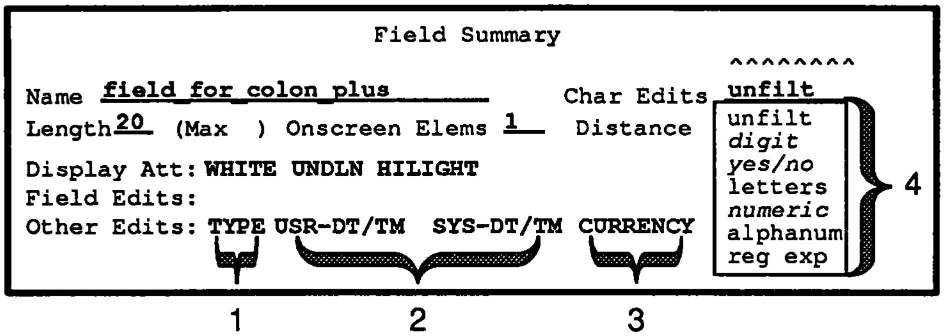

Figure 21: Null field edit window in JAM Screen Editor.

If the user or program does not enter text in the field named `address`, the field is null and JAM will display the string, `*******` as the field contents. JAM/DB*i* would convert the string `*******` to `NULL` (i.e., the value of the engine's null string) before passing it to a DBMS.

If the variable does not have a null edit, or its value does not equal its null edit string, the processor calls a routine to examine field characteristics and determine the variable's JAM *type*.

3. Note that when JAM is executing a *screen entry function*, JAM by default will search for ***variable*** in the LDB before searching the current screen.

A field or LDB variable has exactly one JAM type. Since a variable may have more than one of the qualifying PF4 characteristics, JAM uses some precedence rules when assigning the JAM type.




| Summary Keyword | Setting of Field Characteristic (PF4 menu in draw mode) | Submenu Option | JAM Type |
|---|---|---|---|
| TYPE | type<br>(C types for structures) | char string<br>int<br>unsigned int<br>short int<br>long int<br>float<br>double<br>zoned dec.<br>packed dec. | FT_CHAR<br>FT_INT<br>FT_UNSIGNED<br>FT_SHORT<br>FT_LONG<br>FT_FLOAT<br>FT_DOUBLE<br>FT_ZONED<br>FT_PACKED |
| USR-DT/TM<br>SYS-DT/TM | misc. edits | date or time | DT_DATETIME |
| CURRENCY | misc. edits | currency | DT_CURRENCY |
| Char Edits | char edits | digits only<br>yes/no field<br>numeric | FT_UNSIGNED<br>DT_YESNO<br>FT_DOUBLE |

Figure 22: Field Summary Screen (PF5 in draw mode). Use the summary screen to determine a field's JAM type. A TYPE edit has the highest priority, then a date time edit, then a currency edit, and finally a character edit. A variable with any other edits has the JAM type FT_CHAR.

C record types are assigned with the `type` option on the PF4 key menu. For clarity, we call these types *C types*. To assist developers using utilities such as `f2struct`, JAM automatically assigns a default C type to each field. Developers may also explicitly set a C type. JAM/DB*i* ignores C types assigned by default; it only uses those assigned explicitly by a developer. The field summary screen is an easy way of checking whether or not JAM/DB*i* will use the variable's C type. If the word `TYPE` is shown on the `Other Edits` line of the field summary window, and the type is not `omit`, JAM/DB*i* will use it to assign a JAM type.

Otherwise, JAM examines the miscellaneous edits; a date-time or currency edit will provide a JAM type. If the variable does not have a date-time or currency edit, JAM examines the variable's PF4 `char edits`. An edit of `digits only`, `yes/no field`, or `numeric` will provide a JAM type. For all other field and LDB variables, and for all JPL variables, JAM assigns `FT_CHAR` as the JAM type.

Beware of C type edits that may conflict with other edits. For example, if a field had a type edit `int` and a `date-time edit`, its JAM type would be `FT_INT`. The Screen Manager would enforce the date-time format for user entry but JAM/DB*i* would not convert the date-time string into a format the engine would recognize.

Note: developers may also use `sm_ftype` to determine a variable's JAM type. The assignments are the same as those in the table above, except for JPL variables. The library function `sm_ftype` returns 0, not `FT_CHAR`, for JPL variables.

## Step 3. Format a Non-null Value

Once JAM/DB*i* determines a variable's JAM type, it uses the classification to perform any necessary formatting and returns the formatted text to JPL.

### `DT_DATETIME` Variable

If JAM type is `DT_DATETIME`, the processor calls the support routine to format the text in the engine's default syntax for dates. Some support routines store a JAM date-time format string in the style of the engine. When formatting a field value, it may simply pass the format string and value to JAM's date-time routines to reformat the string. Other support routine may call a conversion function from the DBMS library to perform the task.

Of course, the actual result is dependent on the engine. For example, if the value in a date-time field is December 31, 1999 3:05 PM and the current engine is using the ORACLE support routine, JAM/DB*i* formats the date as

```
TO_DATE('31121999 150500', 'ddmmyyyy hh24miss')
```

If the engine is using the SYBASE support routine, however, JAM/DB*i* formats the date as

```
'Dec 31,1999 3:5:0:000PM'
```

Some engines support more than one datatype for date-time columns. Please see the engine-specific *Notes*.

### FT_CHAR Variables

If JAM type is FT_CHAR, the processor checks if the engine uses quote and escape characters. By default, an engine uses a single quote for quote_char, and a single quote for escape_char.

The processor first determines the size of the formatted text by adding the length of the unformatted text, the number of embedded quote_char's in the text, and 2 (for the enclosing quote characters). If it cannot allocate a buffer large enough for the text, the processor returns the SM_MALLOC error. If the allocation is successful, the processor writes the formatted text to the buffer. It puts a quote_char at the first position in the buffer and, as it copies each character from the source string to the buffer, it compares the character with quote_char. If the character equals quote_char the processor puts an escape_char before the embedded quote_char. A final enclosing quote_char is put at the end of the text.

For example, JAM/DB*i* would format the field value

```
Ms. Penelope O'Brien
```

to

```
'Ms. Penelope O''Brien'
```

JAM/DB*i* would format the field value

```
Reported record sales for "The Novice's Guide to PC's"
```

to

```
'Reported record sales for "The Novice''s Guide to PC''s"
```

A few engines do not support both single and double quotes within a character string. For engine-specific information, please see the *Notes* section in this document.

### FT_ numeric and DT_CURRENCY Variables

For the remaining JAM types, the processor calls the JAM function sm_strip_amt_ptr to strip editing characters from the numerical string. The function strips all non-digit characters except for an optional leading negative sign and a decimal point. See the JAM *Programmer's Guide* for more information on sm_strip_amt_ptr. The colon preprocessor does not use precision edits when formatting numeric values.

For example, JAM/DB*i* would format

```
$500,000.00
```

as

```
500000.00
```

JAM/DB*i* would format

```
(-89.003)
```

as

```
-89.003
```

It would format

```
001-02-0003
```

as

```
001020003
```

If you wish to preserve embedded punctuation in numeric fields, set the field's C type to `char`.

See the engine-specific *Notes* for additional information.

8.1.2.
## Colon-equal Processing

To specify a NULL value in a search criteria, most engines require the syntax

SELECT ***column_list*** FROM ***table*** WHERE ***column*** IS NULL

To permit endusers to select rows where a column value is either known or unknown (i.e., NULL), use the colon-equal processor. For example,

```
sql SELECT * FROM emp WHERE zip :=zip
```

If `zip` is a character field with the null edit

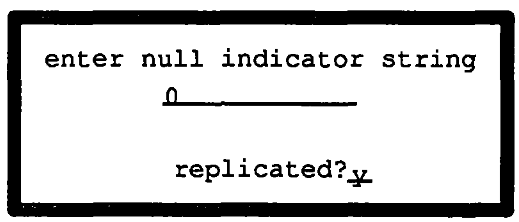



Figure 23: Null field edit window in JAM Screen Editor.

JAM/DB*i* would format the value

```
10038
```

as

```
= '10038'
```

thus executing

```
SELECT * FROM emp WHERE zip = '10038'
```

It would format the field's "null" value

```
00000
```

as

```
IS NULL
```

thus executing

```
SELECT * FROM emp WHERE zip IS NULL
```

8.1.3.

# Examples

## A Field with Default Characteristics

If the current screen has a field named `last` with no field, miscellaneous or type edits, and a character edit `unfilt`, its field summary screen would appear as

```
                              Field Summary
                                                        ^^^^^^^^
Name last                                  Char Edits unfilt
Length20 (Max  ) Onscreen Elems 1  Distance  (Max Occurs  )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits:
```

Figure 24: Field Summary Screen. With these edits, **JAM** type = `FT_CHAR`.

Since the field does not have any of the field characteristics listed in Figure 22 on page 65, **JAM** type = `FT_CHAR`. If the field `last` contained the text `D'Angelo` when the following were executed,

```
sql SELECT * FROM employee WHERE last = :+last
```

JAM/DB*i* would pass the query

```
SELECT * FROM employee WHERE last = 'D''Angelo'
```

If the field `last` were empty, JAM/DB*i* would pass the empty string, not the null string,

```
SELECT * FROM employee WHERE last = ''
```

Null conversion is performed only on variables with a null field edit.

## A Variable with a Date-time Edit and a Null Edit

If the current screen contains a field `hiredate` with a null field edit string `00/00/00`, a date-time edit `MON2/DATE2/YR2` for a user-specified date, and character edit of `digits only`, its summary screen would appear as

```
                              Field Summary
                                                               ^^^^^^^^
Name hiredate                              Char Edits digit
Length 8  (Max   ) Onscreen Elems 1   Distance   (Max Occurs   )

Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: USR-DT/TM NULL
```

Figure 25: Field Summary Screen. For this field, JAM type = `DT_DATETIME`.

Assume that back slash characters are saved with the field as embedded punctuation. Since a date-time edit has a higher precedence than a character edit, the JAM type for this field is `DT_DATETIME`. If the user entered the date `12/31/91` and executed the following function,

```
sql WITH CONNECTION oracle_conn \
   INSERT INTO employee (last, hiredate) \
      VALUES (:+last, :+hiredate)
```

and the engine, for example, were ORACLE, JAM/DB*i* would pass the statement

```
INSERT INTO employee (last, hiredate) VALUES \
      ('D''Angelo', \
       TO_DATE('31121991 000000', 'ddmmyyyy hh24miss'))
```

to the engine.

If the user did not change the text in the field `hiredate`, so that its contents were `00/00/00`, JAM/DB*i* would pass the statement

```
INSERT INTO employee (last, hiredate) \
   VALUES ('D''Angelo', NULL)
```

to the engine.

## A Variable with a Digits Only Character Edit and a C-Type char string Edit

Very often it is useful to use the `digits only` character edit on fields that accept values such as a social security number, zip code, or telephone number. If this is the only edit on the field, the colon-plus processor will format the field's value as an unsigned integer, removing embedded punctuation and leading zeros. However, if the developer resets the C-type edit to `char string`, the colon-plus processor will format the field's contents as a character string, preserving embedded punctuation and leading zeros.

If the current screen contains a field `zip_code` with a character edit of `digits only` and a C type of `char string`, its summary screen would appear as

```
                          Field Summary
                                                    ^^^^^^^^
Name zip code                          Char Edits  digit
Length5  (Max  ) Onscreen Elems 1      Distance    (Max Occurs  )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: TYPE
```

Figure 26: Field Summary Screen. For this field, JAM type is set according to the value of TYPE. If TYPE is "char string" JAM type = `FT_CHAR`.

For example, if a user entered `00912` in the field `zip_code` and executed the following function,

```
sql SELECT * FROM marketing WHERE zip = :+zip_code
```

JAM/DB*i* would pass the query

```
SELECT * FROM marketing WHERE zip = '00912'
```

to the DBMS.

Note that if the developer assigned `digit only` to the field, but did not reset the C type, JAM/DB*i* would pass the query

```
SELECT * FROM marketing WHERE zip = 912
```

# 8.2.
# USING PARAMETERS IN A CURSOR DECLARATION

Some engines permit parameters in the SQL statement of a cursor declaration statement. Therefore, they permit one or more values to be supplied when the cursor is executed. On those engines that do not support binding (e.g., Progress and SYBASE) JAM/DB*i* internally supports cursors with parameters.

When JAM/DB*i* executes a DECLARE CURSOR statement, it scans the statement for parameters. For all engines, JAM/DB*i* recognizes

: : ***parameter***

to be a parameter.[4] If JAM/DB*i* finds a parameter, it sets up a data structure for it. It will attempt to find a value for the parameter when the cursor is executed. Parameters may be used to supply column values for any SELECT, INSERT, UPDATE, or DELETE statement. For example,

```
dbms DECLARE a_cursor CURSOR FOR \
   SELECT * FROM emp WHERE last = ::xyz

dbms DECLARE b_cursor CURSOR FOR \
   INSERT INTO acc VALUES (::ss, ::sal, ::exmp)

dbms DECLARE c_cursor CURSOR FOR \
   UPDATE emp SET street=::street, city=::city, \
   st=::st, zip=::zip WHERE ss=::ss

dbms DECLARE d_cursor CURSOR FOR \
   DELETE newsales WHERE custid=::id
```

The binding data structures are stored with an individual cursor. Therefore, the application should give a unique name to each parameter belonging to a single cursor. A cursor cannot have two parameters with the same name.

4. Many vendors use a single colon to begin a parameter name. Since this form conflicts with the colon preprocessor, two colons must be used in JPL. The second colon prevents the colon processor from performing variable substitution. Some vendors, such as INFORMIX, use a single question mark to represent a parameter. JAM/DB*i* also recognizes these engine-specific forms.

A value for a parameter is supplied in the USING clause of an EXECUTE statement,

```
dbms WITH CURSOR cursor EXECUTE USING arg [, arg...]
```

JAM/DB*i* looks for the keyword USING before passing the cursor's query to the DBMS. If it finds the keyword, it assumes the arguments which follow are parameter values. If an ***arg*** is not quoted, JAM/DB*i* assumes it is a variable and performs variable substitution and formatting. Values and parameters may be bound by position. For example,

```
dbms DECLARE b_cursor CURSOR FOR \
    INSERT INTO acc VALUES (::p1, ::p2, ::p3)
....
dbms WITH CURSOR b_cursor EXECUTE USING ss, sal, exmp
```

Values and parameters may also be bound explicitly by name,

```
dbms DECLARE b_cursor CURSOR FOR \
    INSERT INTO acc VALUES (::p1, ::p2, ::p3)
....
dbms WITH CURSOR b_cursor EXECUTE \
    USING p3=exmp, p1=ss, p2=sal
```

Note that p3, p1, and p2 are not JAM variables but exmp, ss, and sal are. JAM/DB*i* uses the values of exmp, ss, and sal to execute the INSERT. To supply a literal value to the INSERT, put the value in quotes,

```
dbms WITH CURSOR b_cursor EXECUTE \
    USING p1=ss, p2=sal, p3="0"
```

JAM/DB*i* formats binding values in a method similar to the colon-plus processor. This is discussed in detail in the next section.

On those engines that support parameters, using them often improves the efficiency of the application, especially when a query is executed several times. On engines where JAM/DB*i* simulates support, such as SYBASE, the use of parameters will be less efficient. However, the convenience and the greater ease of portability may compensate for the additional processing.

8.2.1.

## Parameter Substitution and Formatting

An ***arg*** in a USING clause may be either

- a quoted string, or
- a JAM variable

Colon-plus processing is not necessary because JAM/DB*i* automatically formats the value of parameter variables. If the variable is an array name, an occurrence number may be given. If no occurrence is given, JAM/DB*i* concatenates all the non-empty occurrences in the array, separating the occurrences with a single space. Substrings are not permitted.

For each cursor, JAM/DB*i* maintains binding information. When a cursor's statement uses parameters, JAM/DB*i* stores the names of the parameters. When a cursor is executed, JAM/DB*i* compares the values in the DBMS EXECUTE statement with the binding information from the cursor's declaration. This permits both positional and explicit binding.

JAM/DB*i* uses a data structure to store the formatted text and JAM type of ***arg***. If ***arg*** is not quoted, JAM/DB*i* assumes it is a variable and calls `sm_ftype` to determine the variable's `ftype` code and flags. Like the colon-plus processor, the binding routine distinguishes between empty and null variables; a variable is null if it has a null edit and contains the null edit string.

If `ftype=DT_DATETIME`, JAM/DB*i* calls the support routine to convert the value to a binary date-time value. See the discussion of DT_DATETIME on page 66 for more information.

No processing is done on the values of `FT_CHAR` variables or quoted strings.

For all other types, JAM/DB*i* strips characters other than digits, the decimal point, and a leading negative sign from the value.

Below are some examples showing the different formats for ***arg*** in a USING clause.

```
dbms DECLARE x CURSOR FOR \
   SELECT * FROM emp WHERE name=::p1 or ss=:p2

# newname and ss_number are LDB variables
dbms WITH CURSOR x EXECUTE \
   USING p1=newname, p2=ss_number

# code is a JPL variable containing the text "ss_number"
# and ss_number is a field on the current screen
dbms WITH CURSOR x EXECUTE USING p1='Jones', p2=:code

# name and ss_number are field arrays. i is a JPL variable
dbms WITH CURSOR x EXECUTE \
   USING p1=name[i], p2=ss_number[i]
```

## Examples

If the current screen contained a field named `total` with a currency edit and character edit of `numeric` its summary screen would appear as

```
                         Field Summary
                                                  ^^^^^^^^
Name total____________________________    Char Edits numeric_
Length15_ (Max  ) Onscreen Elems 1__  Distance   (Max Occurs  )

Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: CURRENCY
```

Figure 27: Field Summary Screen. For this field, `ftype = DT_CURRENCY`.

If the user entered the total `$9,499.99` and executed the following statements:

```
dbms DECLARE sales_cursor CURSOR FOR \
   SELECT * FROM orders WHERE total > ::x
...
dbms WITH CURSOR sales_cursor EXECUTE USING x=total
```

the DBMS would execute

```
SELECT * FROM orders WHERE total > 9499.99
```

If the current screen contained a field named `description` with a `null field` edit and a `word wrap` edit, its summary screen would appear as

```
                         Field Summary
                                                  ^^^^^^^^
Name description______________________    Char Edits unfilt__
Length35_ (Max  ) Onscreen Elems 5__  Distance   (Max Occurs10)

Display Att: WHITE UNDLN HILIGHT
Field Edits: WDWRP
Other Edits: NULL
```

Figure 28: Field Summary Screen. With these edits, `ftype = FT_CHAR`.

If the user executed the following statements:

```
dbms DECLARE ins_cursor CURSOR FOR \
   INSERT INTO products (description) VALUES (::p1)
...
dbms WITH CURSOR ins_cursor EXECUTE USING description
```

when the word wrapped array were empty, the DBMS would execute

```
INSERT INTO products (description) VALUES ('')
```

If, however, the array contained text, JAM/DB*i* would concatenate the non-empty occurrences into one long string which the DBMS would insert into the column `description`.

# Chapter 9. Data Flow from a Database

A JAM/DB*i* application receives two types of information from a database:

- data requested by a SELECT statement
- a count of the rows fetched for a SELECT statement
- error and status codes from an engine and from JAM/DB*i*

The rest of the chapter discusses how this information flows from one or more databases to variables in a JAM application. The first part discusses the destination and format of data returned by SELECT statements. The second part discusses the global JAM/DB*i* variables for status and error data.

In addition to the two types of information described above, an application may also receive data as the result of executing a stored procedure. Since all engines do not support stored procedures, and the syntax of commands varies among those that do, the topic is covered in the *Notes* section of this document.

## 9.1.
# DATA FETCHED BY SELECT

When a SELECT statement is passed to an engine, JAM/DB*i* performs several steps before transferring data to JAM variables.

1. JAM/DB*i* counts the number of columns in the query and records information on each column's name, length, and type. Type is DT_DATETIME, FT_INT, or FT_CHAR.
2. For each column, it searches for a JAM variable destination. If a destination exists, JAM/DB*i* records the length of the variable. If no JAM destination exists for a column, or the destination is an LDB constant, JAM/DB*i* does no fetches for the column. The discussion of JAM destinations is in Section 9.1.1. on page 78.
3. It determines the number of rows to fetch. This number usually equals the number of occurrences in the smallest JAM destination variable, or 0 if there are no target variables. See Section 9.1.2. on page 83.
4. Finally, JAM/DB*i* formats data before writing it to the destination variables if the database column has a date datatype, or if the destination variable has a null, currency, or precision edit. See Section 9.1.3. on page 89.

The sequence above describes a SELECT that writes database column values to individual occurrences of a field, JPL variable, or LDB variable. Developers may also direct the results of a SELECT to two other types of targets. See Section 9.1.4. on page 92 for more information.

### 9.1.1.
## JAM Targets for a SELECT

For an application to retrieve data from a database, there must be an unambiguous mapping between a selected database column and its JAM destination. There are two ways of associating JAM targets with database columns.

- The developer gives a JAM target variable the same name as a database column. This is called *automatic mapping*.
- The developer explicitly declares a JAM variable as the target of a database column. This is called *aliasing*.

## Automatic Mapping

By default when executing a SELECT statement, **JAM**/DB*i* will search for **JAM** variables with the same names as the specified columns. For the statement,

```
sql SELECT lastname, ssnumber, dept, date FROM emp
```

to return values to **JAM** variables, the table emp must have at least four columns: `lastname`, `ssnumber`, `dept`, and `date`. If any of these columns does not exist in the table emp, the engine returns an error.

The application may have a **JAM** destination variable for none, some, or every named column in the SELECT statement. To return the values of all four columns to the application, then there must be a **JAM** variable for each column. The variables may be named `lastname`, `ssnumber`, `dept`, and `date`. If one of these fields does not exist, **JAM**/DB*i* ignores the values belonging to that particular column.

Developers may also use one or more qualified column names in SELECT statements. For example,

```
sql SELECT emp.lastname, emp.ssnumber, emp.dept, \
   emp.date FROM emp
```

The **JAM** targets, however, must be given unqualified names: `lastname`, `ssnumber`, `dept`, and `date`.

**JAM**/DB*i* also permits the use of the shortcut SELECT statement,

```
sql SELECT * FROM emp
```

Using automatic mapping, **JAM**/DB*i* looks for a **JAM** variable for each column in the table emp. Columns without matching variables are simply ignored. This is not treated as an error.

When using automatic mapping, the case of the **JAM** variable names should correspond to the case flag used in the engine initialization in `dbiinit.c`. If the engine's case flag is `DM_FORCE_TO_LOWER_CASE`, the **JAM** variables for a SELECT should have lower case names. If the case flag is `DM_FORCE_TO_UPPER_CASE`, the **JAM** variables should have upper case names. If the case flag is `DM_PRESERVE_CASE`, the **JAM** variables should use the exact case of the database columns.

## Aliasing

Aliasing is used when automatic mapping is inconvenient or impossible to use. In particular, aliasing is necessary when selecting any of the following:

- a column whose name is not a legal **JAM** variable name

- a column whose name conflicts with other JAM variable names in the application
- a computed column or an aggregate function (COUNT, SUM, AVG, MAX, MIN)

Aliasing is not limited to these conditions. Any or all columns may be aliased if desired. Occasionally, developers like to alias a column if its name is not descriptive or because they wish to name target variables for a particular table and column.

Developers use the command DBMS ALIAS to specify aliases. On some engines, developers may also use the engine's SELECT syntax to specify aliases.

### Using DBMS ALIAS

DBMS ALIAS is associated with a SELECT cursor, either a named cursor or the default SELECT cursor. If a cursor is not named, the aliases affect all SELECT's executed with the default cursor. The syntax for assigning aliases to a cursor is either of the following:

```
dbms [WITH CURSOR cursor] ALIAS column1 jam_var1 \
    [, column2 jam_var2 ...]
```

to alias a column name to a JAM variable, or

```
dbms [WITH CURSOR cursor] ALIAS [jam_var1] \
    [, [jam_var2] ...]
```

to alias a column position to a JAM variable. Either named or positional aliasing may be used, but both forms may not be used in a single statement.

To turn off aliasing, execute DBMS ALIAS without any arguments. Again, if a cursor name is given, aliasing is turned off on the named cursor. If no cursor name is given, aliasing is turned off on the default cursor.

The case of the column names in the DBMS ALIAS statement should correspond to the case flag used in the engine initialization in dbiinit.c. If the engine's case flag is DM_FORCE_TO_LOWER_CASE, the column names should be in lower case. If the case flag is DM_FORCE_TO_UPPER_CASE, the column names should be upper case. If the case flag is DM_PRESERVE_CASE, the column names should use the exact case of the database columns. The case of *jam_var* should always match the exact case of the JAM variable name.

If an application aliases a column to a JAM variable that does not exist JAM/DB*i* ignores the column's values. This is NOT treated as an error.

### Using DBMS ALIAS to Alias Column Names

First consider an example that aliases column names to JAM variables. For example,

```
dbms ALIAS first firstname, last lastname
sql SELECT ssn, last, first FROM emp
```

JAM/DB*i* writes the values from the column `first` to the variable `firstname` and it writes the values of column `last` to the variable `lastname`. Since no alias was given for `ssn`, it maps it to a variable of the same name. See the figure below.

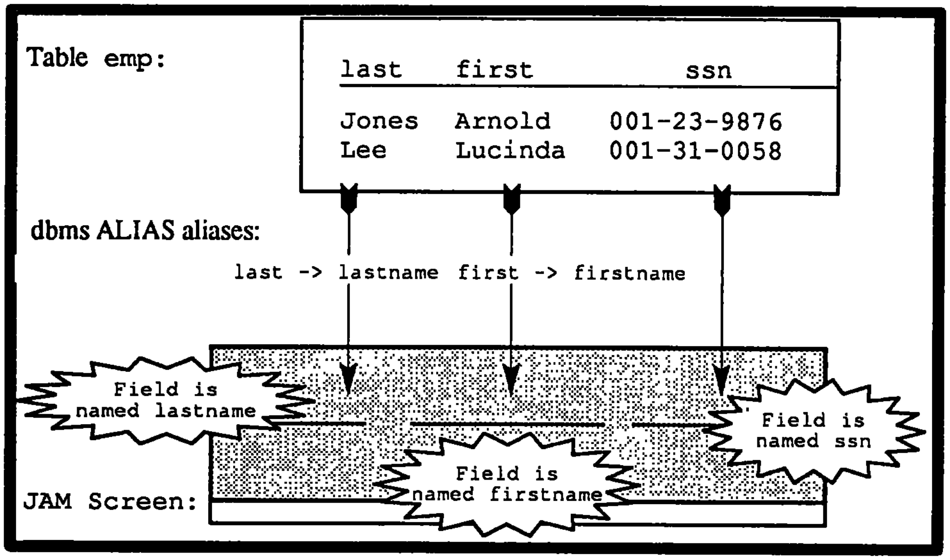



Figure 29: The mapping of `SELECT ssn, last, first FROM emp` when aliases are used.

Aliases may also be given after declaring a named cursor. For example,

```
dbms DECLARE sales_cursor CURSOR FOR \
    SELECT inv#, sale_date, ship_date, amount FROM acc
dbms WITH CURSOR acc_cursor ALIAS "inv#" invoice_id
dbms WITH CURSOR acc_cursor EXECUTE
```

Since `inv#` is not a legal JAM variable name, the application must declare an alias for the column if it is to receive the column's value. Before executing the cursor, the application aliases column `inv#` to variable `invoice_id`. The cursor keeps this alias until the application turns it off with DBMS ALIAS or closes the cursor with DBMS CLOSE CURSOR. If a column name is not a valid JAM identifier, enclose it in quote characters; this ensures that JAM/DB*i* parses it correctly.

## Using DBMS ALIAS to Alias Column Positions

Now consider an example that uses positional aliases. For example,

```
dbms ALIAS min_salary, max_salary, avg_salary
sql SELECT MIN(sal), MAX(sal), AVG(sal) FROM acc
```

JAM/DB*i* writes the aggregate function values to the alias variables. `MIN(sal)` is written to the variable `min_salary`, `MAX(sal)` is written to the variable `max_salary`, and `AVG(sal)` is written to the variable `avg_salary`. Note that there is no automatic mapping available. If the application had not declared aliases, the values would not be written to JAM variables.

Of course, the application should turn off the positional aliases when it is finished. If it does not turn them off before executing the next SELECT, JAM/DB*i* will attempt to write the first three columns' value to the three positional alias variables. If those variables are no longer available, JAM/DB*i* will ignore the first three columns in the SELECT set.

### Using the Engine's SELECT Syntax

Many engines support aliasing in their SELECT syntax. In interactive mode, this permits the user to specify for a view a column heading that is different than the database column name. Typically, the syntax is

SELECT ***column1 heading1*, *column2 heading2***...FROM ***table***

In interactive mode, the values of ***column1*** are placed under the heading ***heading1***, and the values of ***column2*** are places under the heading ***heading2***. Please note that in this syntax a space separates a column from its alias, and a comma separates the column-alias set from the next column or column-alias set. Some engines may support another syntax. See your database documentation for details.

If an engine supports aliasing in a SELECT statement, JAM/DB*i* will also support it. Developers may follow the syntax of the engine, replacing ***heading*** with the name of the appropriate JAM variable.

For example, if the syntax shown above is supported by the engine, than the following could be used in a JAM/DB*i* application,

```
sql SELECT id product_no, supplier, ucost price FROM inv
```

When this statement is executed, the DBMS tells JAM/DB*i* that the columns `product_no`, `supplier`, and `ucost` were selected. JAM/DB*i* will look for variables with those names. If there is a variable `id` available, this SELECT statement will not write to it because the engine has aliased it to `product_no`.

Although this form is supported, we recommend the use of DBMS ALIAS, especially for applications accessing more than one engine. JAM/DB*i* provides identical support for DBMS ALIAS on all engines.

9.1.2.
# Number of Rows Fetched

A SELECT set often contains more than one row. JAM/DB*i* must determine how many rows it may fetch at one time from a SELECT set. The rest of the SELECT is fetched by executing one or more DBMS CONTINUE's.

- If an occurrence number was specified with a target variable name, only one row is fetched.
- If a target is a word wrapped array, only one row is fetched.
- If using browse mode, only one row is fetched. (See the engine-specific *Notes*).

Otherwise, JAM/DB*i* examines the number of occurrences in each of the targeted variables. Usually, all the target variables have the same number of occurrences. If this is true, JAM/DB*i* fetches a row for each occurrence. If the targets do not have the same number of occurrences, JAM/DB*i* finds the target variable with the least number of occurrences and fetches that number of rows. Be careful of LDB variables that are unintentional targets of a SELECT especially when using the wild card * in a SELECT or when executing a SELECT in a screen entry function.

For example, consider an application using the wild card,

```
sql SELECT * FROM table
```

The application has onscreen fields for some of the columns in the table. The LDB, however, contains an entry with the name of one of these unrepresented columns. If the onscreen fields have 20 occurrences and the LDB entry has 5 occurrences, the SELECT will fetch only five rows at a time.

Also, consider an application that executes a SELECT in a screen entry function. By default, JAM first searches the LDB and then the screen for JAM variables when executing screen entry functions. Therefore, if a variable is represented both as an onscreen field and as an LDB variable, a screen entry function will write to the LDB variable before the LDB merge writes to the onscreen field. If the LDB variable and the field do not have the same number of occurrences, data is lost or appears lost when the LDB merge updates the screen fields.

# Scrolling Through a SELECT Set

Most JAM/DB*i* developers must create applications capable of handling a fluctuating number of data rows. Based on the type of data selected and the hardware in use, a developer may use either or both types of scrolling—JAM scrolling or JAM/DB*i* scrolling.

With JAM scrolling, the application uses large scrolling arrays as the destination variables of a SELECT statement. The entire SELECT set is fetched in a single step and the user presses the page up and page down keys (logical keys SPGU and SPGD) to view the rows.

With JAM/DB*i* scrolling, the application uses single-element fields or non-scrolling arrays as the destination variables of a SELECT statement. The SELECT set is fetched incrementally. To permit the user to scroll backward and forward in the set, the application must set up function keys to execute the JAM/DB*i* scrolling commands.

The two methods are described in detail below.

### JAM-based Scrolling

JAM-based scrolling is useful for small to mid-sized SELECT sets. The upper limit on the number of rows is 9999, the maximum number of occurrences allowed for a JAM variable. Since the application must keep the entire SELECT set in memory, the realistic limit may be much lower on a platform like MS-DOS or for a SELECT involving many columns.

With this approach, the developer creates large scrolling arrays with more occurrences than the number of rows he or she expects to be in the SELECT set. When the SELECT is executed at runtime, there is no penalty for unused occurrences; JAM allocates only whatever memory is needed to hold the returned rows. Therefore, a JAM screen might contain variables each with 10 elements and 1000 occurrences. If a SELECT set contained only 75 rows JAM would allocate memory for 75 occurrences in each of the variables; it would not allocate memory for the 925 unused occurrences.

There are several ways of verifying that the arrays actually contained enough occurrences to hold the entire SELECTset. Most often the application examines the value of the global variable `@dmretcode`. JAM/DB*i* writes a no-more-rows status code to this variable when the engine signals that it has returned all requested rows. The value of this variable may be examined after a SELECT. See page 93 for more information on these variables. An example procedure is shown below:

```
proc select_all
# DM_NO_MORE_ROWS is an LDB constant.
sql SELECT inv_no, prod_no, prod_desc, quantity, \
   unit_price, total FROM new_sales
if @dmretcode == DM_NO_MORE_ROWS
   msg esmg "All rows returned."
else
   msg emsg "Application could not display all orders."
return
```

This approach is very easy to use. Since all the rows are fetched at once, the application makes only one request of the database server and it is free to use the default SELECT cursor to make new selects.

It is not the best method for large SELECT sets. If the application is too slow displaying the data or is sluggish after the rows have been fetched, the developer should consider JAM/DB*i*-based scrolling or some other alternative scroll driver.

### JAM/DB*i*-based Scrolling

JAM/DB*i*-based scrolling is useful for mid-sized to large SELECT sets. Neither **JAM** nor **JAM/DB*i*** impose any limit on the number of rows that may be displayed with this method.[5]

With this approach, developers create non-scrolling arrays. The target fields contain elements to display one or more rows on the screen at time. At least two procedures are needed to view the SELECT set. The first procedure executes the SELECT and fetches the first screenful of rows. The second procedure executes a DBMS CONTINUE to fetch the next screenful of rows from the SELECT set. The second procedure may be executed many times before the user sees all the rows.

For example, the current screen has fields named for the columns in the table emp. Each field has five elements. The application uses the procedures like the following to select data from a table:

```
proc select_emp
sql SELECT * FROM emp
return

proc continue_select
dbms CONTINUE
return
```

as well as control strings like the following:

```
PF1    ^jpl select_emp
PF2    ^jpl continue_select
```

Assume that table emp contains 12 rows. When the user presses the `PF1` key, the application executes the JPL procedure `select_emp` and writes rows 1 through 5 to the screen. If the user presses `PF2`, the application executes the procedure `continue_select` which clears the arrays and writes rows 6 through 10 to the screen. If the user presses `PF2` again, the application executes `continue_select` again which clears the arrays and writes rows 11 and 12 to the screen. If the user presses `PF2` a third time, the application does nothing because there are no more rows in the SELECT set.

An application may simulate scrolling through a SELECT set by using the following commands:

5. In multi-user environments developers should know how the engine ensures read consistency: the guarantee that data seen by a SELECT does not change during statement execution. The engine may be using rollback segments or shared locks to provide read consistency. Since a shared lock prevents other users from updating locked rows, applications on these engines should release the lock as soon as possible. See the engine-specific *Notes* for more information.

- DBMS CONTINUE_UP — to scroll up a screenful of rows
- DBMS CONTINUE_TOP — to scroll to the first screenful of rows
- DBMS CONTINUE_BOTTOM — to scroll to the last screenful of rows

Some engines have native support for these commands. For example, the engine may buffer the rows in memory on the server. JAM/DB*i* also provides its own support for these commands. Applications may use DBMS STORE FILE to set up a continuation file for a named or default SELECT cursor. When it is used, JAM/DB*i* buffers SELECT rows in a temporary binary file. The syntax of the command is

```
dbms [WITH CURSOR cursor] STORE FILE [file]
```

The command is supported on all engines. To select and view data, an application uses procedures like the following:

```
proc select_emp
dbms STORE FILE
sql SELECT * FROM emp
return

proc scroll_down
dbms CONTINUE
return

proc scroll_up
dbms CONTINUE_UP
return

proc scroll_top
dbms CONTINUE_TOP
return

proc scroll_end
dbms CONTINUE_BOTTOM
return
```

as well as control strings like the following:

```
PF1    ^jpl select_emp
PF2    ^jpl scroll_down
PF3    ^jpl scroll_up
PF4    ^jpl scroll_top
PF5    ^jpl scroll_end
```

Using the same number of rows and occurrences as earlier, when the user presses the PF1 key, the application executes the JPL procedure `select_emp` and writes rows 1 through

5 to the screen. If the user presses PF2, the application executes the procedure scroll_down which clears the arrays and writes rows 6 through 10 to the screen. If the user presses PF3, the application executes scroll_up which clears the arrays and writes rows 1 through 5 to the screen. If the user presses PF5 the application executes scroll_end which clears the arrays and writes the last 5 rows in the SELECT set, rows 8 through 12, to the screen.

Although function keys are needed to call the JPL procedures which execute the **JAM/DB*i*** scrolling commands, end users usually prefer the standard page up and page down keys to the PF keys. The logical keys SPGU and SPGD are not listed in the **JAM** Control String window of the screen editor but their logical values may be reassigned with the **JAM** library function sm_keyoption. Therefore, the application may use an entry and exit function to change how SPGU and SPGD work on a screen or in a field. The entry function calls sm_keyoption so that SPGU acts like the function key that calls the scroll up procedure, and calls sm_keyoption so that SPGD acts like the function key that calls the scroll down procedure. The exit function calls sm_keyoption to restore the default behavior.

Developers who wish to use JPL to call sm_keyoption must install the function in the prototyped list in funclist.c. The JPL procedure must also use the decimal or hexadecimal values of the logical keys. The hexadecimal values are listed in the JAM *Configuration Guide* in the key file chapter. An example function is shown below. This function could be used as the field entry and exit on each target field.

```
vars ENTRY(4) EXIT(4)
vars SPGU(6) SPGD(6) APP1(6) APP2(6) KEY_XLATE(1)
cat ENTRY      '128'
cat EXIT       '16'
cat SPGU       '0x113'
cat SPGD       '0x114'
cat APP1       '0x6102'
cat APP2       '0x6202'
cat KEY_XLATE  '2'

proc entry_exit
parms f_no f_data f_occ f_flag
# APP1    ^jpl scroll_up
# APP2    ^jpl scroll_down
   if (f_flag & ENTRY)
   {
      call sm_keyoption :SPGU :KEY_XLATE :APP1
      call sm_keyoption :SPGD :KEY_XLATE :APP2
   }
   else if (f_flag & EXIT)
```

```
{
    call sm_keyoption :SPGU :KEY_XLATE :SPGU
    call sm_keyoption :SPGD :KEY_XLATE :SPGD
}
return
```

JAM/DB*i*-scrolling uses less memory than JAM scrolling. The application needs only enough memory for the rows displayed on screen. The other rows are buffered either in a binary disk file or by the database server. With large SELECT sets, this approach often improves the application's performance and response time.

This approach requires a little more work by the developer. The application needs procedures to handle the scrolling and possibly the remapping of cursor control keys. Also, the method restricts the SELECT cursor. If the application needs to perform other SELECT statements while scrolling through this set, the application must declare named cursors.

## Controlling the Number of Rows Fetched

Developers using field or LDB arrays as the destinations of a SELECT may specify the maximum number of rows to fetch and the first occurrence to write to in the array destination. The command is

```
dbms [WITH CURSOR cursor] OCCUR int        [MAX int]
dbms [WITH CURSOR cursor] OCCUR CURRENT    [MAX int]
```

See the *Reference Guide* in this document for information.

## Choosing a Starting Row in the SELECT Set

A developer may also change the number of rows fetched by using the command

```
dbms [WITH CURSOR cursor] START int
```

The command tells JAM/DB*i* to read and discard *int* – 1 rows before writing the rest of the SELECT set to JAM variables.

See the *Reference Guide* in this document for information.

9.1.3.

# Format of SELECT Results

Before writing a database column value to a JAM variable occurrence, JAM/DB*i* determines the data type of the database column. In all cases, if the value equals the engine's null (e.g., NULL), JAM/DB*i* writes clears the variable. If the variable has a null field edit, JAM automatically converts the null string to the one assigned by the field edit.

If any value is longer than the variable, the data is truncated.

## Character Column

If a column has a character datatype, the value is simply written to the target variable. If the variable has a word wrap edit or a right-justified edit, the edit is applied.

## Date-time Column

If a column has a date datatype, JAM/DB*i* formats the value before writing it to a JAM variable. If the variable has a date-time edit, JAM/DB*i* uses it. If the variable does not, JAM/DB*i* uses the format assigned to the message file entry `SM_0DEF_DTIME`. By default, the entry is

```
SM_0DEF_DTIME = %m/%d/%2y %h:%0M
```

For example, `April 1, 1991 10:05:03` would be formatted as `4/1/91 10:05`. When the message file default is used, JAM/DB*i* assumes a 12-hour clock.

See the *Author's Guide* and the *Configuration Guide* in the JAM documentation for information on date-time formats.

## Numeric Column

If a column has an integral type, JAM/DB*i* converts the value to a long. JAM then converts the value to ASCII and writes it to the variable, truncating any data longer than the destination field.

If a column has a real type, JAM/DB*i* converts the value to a double. Before writing the value to a JAM variable, JAM/DB*i* determines the precision by examining the variable's currency and/or C type edit.

- *The field has a currency edit, but no C type edit.* If the value is less precise than the edit's minimum number of decimal places, the value is padded to the minimum number of decimal places. If the value is more precise, it is rounded or adjusted to the currency edit's maximum number of decimal places. Note that the round up, round down, or adjust option of the currency edit is applied.

- *The field has a C type edit, but no currency edit.* If the C type is one of the integer types, the value is adjusted by standard rounding to 0 places. If the C type is float or double, the value is padded or adjusted to the type's precision.
- *The field has a currency edit and C type edit that conflict.* If the value is less precise than the currency edit's minimum number of decimal places, the value is padded to the minimum number of decimal places. If the value is more precise than the minimum number of places, JAM/DB*i* compares the currency's maximum number of places and the C type's precision, and uses the less precise of the two. If it uses the currency's maximum number of places, then it also uses the currency's round up, round down, or adjust option. If it uses the C type precision, it adjusts by standard rounding to the precision.
- *The field has neither a currency edit or a C type edit.* The precision defaults to 2.

See the *Author's Guide* in the JAM documentation for more information on currency edits.

## Fetching Unique Column Values

By default, when a column is selected JAM/DB*i* returns all values. JAM/DB*i* also provides a command for displaying only a column's unique values,

```
dbms [WITH CURSOR cursor] UNIQUE column [column ...]
```

JAM/DB*i* replaces a repeating value with the empty string.

This command is useful if an application is selecting values from a table which uses two or more columns as the primary key. For example, if the table `projects` has the columns `project_id`, `staff`, `task_code` and the columns `project_id` and `staff` constitute the primary key, an application could suppress the repeating values in one of the columns of the primary key to improve readability on the screen.

| project_id | staff | task_code |
|---|---|---|
| 1001 | Jones | A |
| 1001 | Carducci | A |
| 1001 | Bryant | C |
| 1004 | Carducci | B |
| 1004 | Mohr | A |
| 1004 | Silver | B |
| 1004 | Thomas | D |
| 1031 | Jones | E |

Figure 30: The primary key of table `projects` is (`project_id, staff`).

```
dbms DECLARE proj_cur CURSOR FOR \
   SELECT * FROM projects ORDER BY project_id
dbms WITH CURSOR proj_cur UNIQUE project_id
dbms WITH CURSOR proj_cur EXECUTE
```

Below is a sample screen displaying the results.

| Project | Employee | Task |
|---|---|---|
| 1001 | Jones | A |
| | Carducci | A |
| | Bryant | C |
| 1004 | Carducci | B |
| | Mohr | A |
| | Silver | B |
| | Thomas | D |
| 1031 | Jones | E |

Figure 31: The **JAM** layout is easier to read than the table layout.

See the *Reference Guide* in this document for more information.

9.1.4.

## Redirecting SELECT Results to Other Targets

Occasionally, developers need other destinations for SELECT statements. JAM/DB*i* provides a feature for concatenating a full result row and writing it to either a JAM variable or a text file.

```
dbms [WITH CURSOR cursor] CATQUERY TO jam_var \
    [SEPARATOR text] [HEADING [ON | OFF] ]
```

```
dbms [WITH CURSOR cursor] CATQUERY TO FILE filename \
    [SEPARATOR text] [HEADING [ON | OFF] ]
```

JAM/DB*i* also provides a command for formatting the results,

```
dbms [WITH CURSOR cursor] FORMAT [column] format
```

See the *Reference Guide* in this document for details.

## 9.2. STATUS AND ERROR CODES

JAM/DB*i* supplies several pre-defined variables where it stores error and status data for the application. These variables are

- `@dmretcode` — The status of the last executed `dbms` or `sql` statement. Its value is 0 or one of the codes defined in `dmerror.h`.
- `@dmretmsg` — A message describing the status of the last executed `dbms` or `sql` statement. Its value is empty or one of the messages from the JAM message file. If `@dmretcode` is 0, this variable is empty.
- `@dmengerrcode` — An engine-specific error code for the last executed `dbms` or `sql` statement. Its value is 0 or an engine-specific code. If 0, the engine did not detect any errors.
- `@dmengerrmsg` — An engine-specific error message for the last executed `dbms` or `sql` statement. If `@dmengerrcode` is empty, this variable is also empty.
- `@dmengwarncode` — An engine-specific warning code or bit setting for the last executed `dbms` or `sql` statement. If empty, the engine did not detect any warning conditions.
- `@dmengwarnmsg` — An engine-specific warning message describing the warning code for the last executed `dbms` or `sql` statement. If `@dmengwarn` is a byte or is blank, this variable is also empty.
- `@dmengreturn` — The return code from the last executed stored procedure. Its value is either blank or an integer. If blank, the engine did not supply a return code.
- `@dmrowcount` — The number of rows fetched to JAM variables by the last SELECT or CONTINUE statement. See the engine-specific *Notes*.
- `@dmserial` — An engine-generated value for a serial column. Its value is 0 or an appropriate serial value for the column. See the engine-specific *Notes*.

After executing a statement JAM/DB*i* updates these variables with any error, warning, or status information returned by the engine. In addition to the engine-specific codes and messages, JAM/DB*i* also supplies engine-independent codes and messages to the variables `@dmretcode` and `@dmretmsg`.

These global variables are available throughout the application from both JPL and C. Note that JAM/DB*i* does not automatically display these values, except in the case of error messages.

JAM/DB*i* uses a default error handler when executing `dbms` and `sql` commands from JPL or C. If a JAM/DB*i* error occurs, the default error handler displays an error message. The source of the message depends on the message flag used to initialize the engine, either the `DM_DEF_ENG_MSG` flag or the `DM_DEF_DBI_MSG` flag.

If a JAM/DB*i* error occurs while executing JPL, the default error handler displays a message and JAM displays the `dbms` or `sql` statement where the error occurred. When the last message is acknowledged, JAM/DB*i* aborts the JPL procedure where the error occurred. An aborted JPL procedure always returns –1 to its caller.

If a JAM/DB*i* error occurs while executing one of the C library functions, the default error handler displays the error message and JAM returns –1 to the function.

An application may override the default handler by installing its own function to handle errors. It may also install an exit function to process all error and status information and display these values to the enduser. This topic is covered in the next chapter.

Chapter 10.

# Hook Functions

JAM/DB*i* provides three hooks for developer-written functions. They are the following

- `ONENTRY` This function is called before executing any `dbms` or `sql` command from JPL or C.
- `ONEXIT` This function is called after executing any `dbms` or `sql` command from JPL or C.
- `ONERROR` This function is called if an error occurs while executing any `dbms` or `sql` command from JPL or C.

JAM/DB*i* hook functions may be written in JPL or C.

A JPL hook function is installed like the following:

```
dbms ONXXXX JPL entry_point
```

where ***entry_point*** is an entry point to a JPL module. An entry point may be a procedure name or a file name. See the JPL Guide for more information.

A C hook function is installed like the following:

```
dbms ONXXXX CALL function
```

where ***function*** is a prototyped function. A prototyped function appears on JAM's `PROTO_FUNC` list. As a JAM/DB*i* hook function, it must be prototyped with three arguments: two strings and an integer. For example,

```
static struct fnc_data pfuncs[] =
{
   {sm_flush()",   flush,    0, 0, 0, 0  },
   ...
   {function(s,s,i)", function, 0, 0, 0, 0 },
}
```

Please consult the JAM *Programmer's Guide* for more information on prototyped functions.

## 10.1. ONENTRY FUNCTION

Before executing a `dbms` or `sql` command from JPL or C, JAM/DB*i* will execute the application's installed ONENTRY function. An ONENTRY function is useful for logging or debugging statements. You may also use an ONENTRY function to modify the JAM environment, for instance remap cursor control keys or change protection edits on fields.

To install an ONENTRY function, use one of the following:

`dbms ONENTRY JPL` ***entry_point***

`dbms ONENTRY CALL` ***function***

To turn off the ONENTRY function, execute the command with no arguments:

```
dbms ONENTRY
```

### 10.1.1. ONENTRY Function Arguments

An ONENTRY hook function receives three arguments:

1. A copy of the first 255 characters of the command line. If the command was executed from JPL, this is the first 255 characters after the JPL command word `dbms` or `sql`.
2. The name of the current engine. If the command used a WITH ENGINE or WITH CONNECTION clause, the argument identifies this engine. If no WITH clause is used, the argument identifies the default engine.
3. A context flag identifying why this function was called. For an ONENTRY function this value is 0.

### 10.1.2. ONENTRY Return Codes

In the present release, the return code from an ONENTRY function is ignored if the current command was executed from JPL. If the command was executed from C, the return code is returned to the calling function.

To ensure compatibility with future releases, it is recommended that this function returns 0.

10.1.3.

## Example ONENTRY Functions

The following sample function logs the current statement in a text file.

```
/* This function is installed as a prototyped function.*/
/* It writes the current time, name of the current */
/* engine, and the command which JAM/DBi will execute */
/* to a file called dbi.log. */

/* dbms ONENTRY CALL dbientry        */

#include "smdefs.h"

int
dbientry (stmt, engine, flag)
char *stmt;
char *engine;
int flag;
{
   FILE *fp;
   time_t timeval;

   fp = fopen ("dbi.log", "a");
   timeval = time(NULL)
   fprintf (fp, "%s\n%s\n%s\n\n",
            ctime(&timeval), engine, stmt);
   fclose (fp);
   return 0;
}
```

This sample function displays a message before performing any JAM/DB*i* operations.

```
# dbms ONENTRY JPL entrymsg

proc entrymsg
   msg setbkstat "Processing. Please be patient..."
   flush
   return 0
```

## 10.2.
# ONEXIT FUNCTION

After executing a dbms or sql command from JPL or C, JAM/DB*i* will execute the application's installed ONEXIT function. An ONEXIT function is useful for logging or debugging statements. You may also use an ONENTRY function to modify the JAM environment, for instance remap cursor control keys or change protection edits on fields. This function is useful for checking error and status codes after each command.

### 10.2.1.
## ONEXIT Function Arguments

An ONEXIT hook function receives three arguments:

1. A copy of the first 255 characters of the command line. If the command was executed from JPL, this is the first 255 characters after the JPL command word dbms or sql.
2. The name of the current engine. If the command used a WITH ENGINE or WITH CONNECTION clause, the argument identifies this engine. If no WITH clause is used, the argument identifies the default engine.
3. A context flag identifying why this function was called. For an ONEXIT function its value is 1.

### 10.2.2.
## ONEXIT Return Codes

The return code from an ONEXIT function is ignored unless an error occurred while executing a sql or dbms command using JPL. If the return code from the function is non-zero, JAM/DB*i* will abort the JPL procedure where the error occurred. If the command is executed from C, the return code is returned to the calling function.

If the application is also using an ONERROR function, the return code from the ONERROR function overrides the return code from the ONEXIT function.

### 10.2.3.
## Example ONEXIT Function

This sample function looks for the no more rows codes after executing a command.

```
# dbms ONEXIT JPL checkstat

# DM_NO_MORE_ROWS is an LDB constant set to 53256

proc checkstat
parms stmt engine flag
if @dmretcode != 0
{
   if @dmretcode == DM_NO_MORE_ROWS
   {
            msg emsg "All rows were returned."
            return 0
   }
   msg emsg "Error executing " stmt "%N" \
            @dmretmsg "%N" @dmengerrrmsg
   return 1
}
return 0
```

## 10.3. ONERROR FUNCTION

If a JAM/DB*i* error occurs while executing a dbms or sql command from JPL or C, JAM/DB*i* will execute the application's installed ONERROR function. An ONEXIT function usually displays the values of the global error variables @dmretmsg and @dmengerrrmsg. It may also display the text of the command that failed. The application may use this function to log error information in a text file.

There are two classes of JAM/DB*i* errors:

- *Syntax or Logic Error in a* dbms *Statement.* Some examples are executing a dbms command that is not supported by the current engine, using an invalid a keyword, executing a cursor that has not been declared, or failing to declare a connection before executing an sql statement. These errors are detected by JAM/DB*i* and reported using standard JAM/DB*i* error codes and messages. These errors update the global variables @dmretcode and @dmretmsg.
- *Engine Error.* Some examples are attempting to SELECT from a non-existent table or column, inserting invalid data in a column, logging on with invalid arguments, or attempting to connect to a server that is not running.

These errors are detected by the engine and reported by the JAM/DB*i* interface. These errors update the global variables `@dmretcode`, `@dmretmsg`, `@dmengerrcode`, `@dmengerrmsg`.

Note that JAM and JPL errors are not a class of JAM/DB*i* errors. In addition to a JAM/DB*i* error, a JPL procedure may fail because of JPL syntax or colon preprocessing errors. If a JPL error occurs, JAM displays an error message describing the error, the source of the JPL statement, and the statement that failed. Furthermore, it aborts the JPL procedure where such an error occurred and returns control to the procedure's caller. It is assumed that JPL and JAM errors are detected and corrected during application development. The only time that developers may need special handling for these errors is during transaction processing. This is discussed in Chapter 11.

An ONERROR function overrides JAM/DB*i*'s default error handler. The function controls the display of error messages. If the error occurred while executing a command from JPL, the ONERROR function also determines whether control is returned to the procedure or to the procedure's caller.

Developers using JPL are encouraged to use an ONERROR function. This ensures consistent error handling throughout the application and reduces the amount of code needed to handle errors. If an ONEXIT function is also installed, JAM/DB*i* calls the ONEXIT function, then the ONERROR function.

To install an ONERROR function, use one of the following:

`dbms ONERROR JPL` ***entry_point***

`dbms ONERROR CALL` ***function***

To turn off the ONERROR function and reinstall the default error handler, execute the command with no arguments:

```
dbms ONERROR
```

10.3.1.
## ONERROR Function Arguments

An ONERROR hook function receives three arguments:

1. A copy of the first 255 characters of the command line. If the command was executed from JPL, this is the first 255 characters after the JPL command word `dbms` or `sql`.
2. The name of the current engine. If the command used a WITH ENGINE or WITH CONNECTION clause, the argument identifies this engine. If no WITH clause is used, the argument identifies the default engine.

3. A context flag identifying why this function was called. For an ONERROR function its value is 2.

10.3.2.

## ONERROR Return Codes

If an application is using an installed error handler, the error handler determines the handling for JAM/DB*i* errors that occur while using JPL.

If a JAM/DB*i* error occurs while executing JPL, a non-zero return code aborts the JPL procedure where the error occurred. The procedure's caller (either JAM or another JPL procedure) gains control. If the return code is 0 however the JPL procedure resumes control; JAM will execute the next statement in the JPL procedure.

If a JAM/DB*i* error occurs while executing C, the ONERROR return code is returned to the calling function.

The return code from an ONERROR function overrides the return code from an ONEXIT function.

10.3.3.

## Example ONERROR Function

```
# DM_ALREADY_ON is an LDB constant.

proc dbi_error_handler
parms stmt engine flag

   if (@dmretcode == DM_ALREADY_ON)
   {
      msg emsg "You are already logged on."
      return 0
   }

   if (@dmengerrcode != 0)
   {
         msg emsg @dmretmsg
         jpl engine_errors :engine
   }
   else
   {
```

```
        msg emsg "Application Error:  " \
           @dmretmsg \
           "See the DBA for assistance."
    }

    return 1

proc engine_errors
parms engine_name
    if engine_name == "xyzdb"
    ...
# Examine DBMS ERROR codes here.
...
```

This procedure first checks if the checks if the error is DM_ALREADY_ON. In this case, it simply displays a message and returns 0. For all other errors, it checks for an engine error code. If there is an engine error it calls another subroutine to check for engine-specific errors. For any other errors, it displays the standard JAM/DB*i* message.

# Chapter 11. Transactions

In addition to the data access capabilities of an engine, JAM/DB*i* supports the engine's transaction processing capabilities.

A transaction is a logical unit of work on a database. The unit of work is usually a set of statements that update a database in a consistent way. That is, the update takes the database from one consistent state to another. Using the familiar personnel database described throughout the document, consider these possible transactions:

- *An employee review transaction.* It involves: an insert to the table `review` supplying a social security number, review date, new salary, and new grade level and an update to the employee's current salary in the table `acc`.
- *A new employee transaction.* It involves: an insert to the table emp supplying the employee's social security number, name, and home address; an insert to the table `review` supplying the employee's social security number, hire date, salary, and grade; and an insert to the table `acc` supplying the employee's social security number, current salary, and number of tax exemptions.

Transaction processing is sometimes a difficult topic for new developers. For one, transaction processing is very engine dependent and thus it requires a clear understanding of the engine's behavior. For another, transaction processing in a JAM/DB*i* application requires careful error processing. For some errors, the application must explicitly tell the engine to undo the transaction. The application must test for these errors.

# 11.1. ENGINE-SPECIFIC BEHAVIOR

As noted earlier, transaction processing is not implemented consistently among SQL databases. Developers should review the documentation on transaction processing supplied by the database vendor before using JAM/DB*i* features.

Generally, transaction processing falls into two types: those that support explicit transactions and those that support auto transactions. An explicit transaction starts with a BEGIN statement; an auto transaction generally starts with the first recoverable statement after a logon, COMMIT, or ROLLBACK. Usually an engine supports either explicit transactions or auto transactions, but not both.

On engines supporting explicit transactions, each COMMIT or ROLLBACK must have a matching BEGIN. On engines supporting autocommit modes, the application may use any number of COMMIT or ROLLBACK statements; if there is no recoverable statement, the COMMI or ROLLBACK is ignored. Engines have different ways of handling transactions that are not terminated by an explicit commit or rollback. Some engines automatically commit or rollback the transaction. Others may leave the database in an inconsistent state. Under no circumstances should the application use the engine's default behavior to terminate a transaction.

The use of explicit rollbacks and commits

- protects the integrity of the database
- makes new and updated data available to the rest of the application and other users at the logical end of the transaction
- releases locks set on tables by the transaction once the transaction is completed, not when the connection closes, permitting the rest of the application or other users to begin new transactions on the tables
- reduces the chances for unrelated operations interfering with one another
- produces applications which are less database-dependent

Finally, although vendors supply commands for transaction processing in their SQL language, developers should use those provided by JAM/DB*i* either with the JPL command `dbms` or the library routine `dm_dbms`. Using `sql` or `dm_sql` to handle transaction processing like commit and rollback is NOT recommended. Using the DBMS versions permits JAM/DB*i* to establish necessary structures and it provides better error handling if a transaction fails.

## 11.2. ERROR PROCESSING FOR A TRANSACTION

The engine is responsible for recovery from system failures such as power loss. Also, if a single statement fails for some reason in the middle of execution, the engine is responsible for rolling back the effects of that statement. If that statement was executed in a transaction, however, the application must execute an explicit rollback to undo any work done between the start of the transaction and the failed statement.

At the very least, JAM/DB*i* must execute a rollback when the engine returns an error to the application. For example, the engine might reject an insert because the row's primary key is not unique. If the insert were part of a transaction, the application should stop executing the transaction and execute a rollback to undo any work done by previous statements in the transaction.

As an additional precaution, developers very likely want to execute a rollback for any error that occurs during the transaction, including an error detected by JAM or JAM/DB*i* before a statement is passed to the engine. An error detected by JAM or JAM/DB*i* rather than the engine is usually the result of a development or maintenance error rather than bad user input (e.g., a statement's colon-plus or binding variable cannot be found because a JAM field was renamed). While these errors should be rare, the application should provide handling for them.

If the transaction processing is done with the JAM/DB*i* C library functions, JAM and JAM/DB*i* error codes are returned to the calling function, either directly or via an installed error handler. If a transaction requires very sophisticated error handling, it may be easier to use these JAM/DB*i* library functions rather than JPL.

If the transaction processing is done in JPL with `dbms`, developers should use the JPL command `retvar` to declare a return variable. A `retvar` variable is set to 0 if a called procedure returns 0 (the default for success) or if a `dbms` or `sql` statement executes without error. If a called procedure aborts because of a JAM error, a `retvar` variable is set to -1. If an installed error handler is called, a `retvar` variable is set to the handler's return code. The JPL Guide in Volume II of the JAM manual has a complete description of this command. The examples in this chapter use `retvar` so that a transaction is rolled back for all JAM/DB*i* and JAM errors.

The best method for transaction processing in JPL uses a generic JPL procedure as a transaction hander. This procedures does the following:

- defines and declares a JPL return variable, ***jpl_retcode***.
- calls a JPL subroutine that contains the actual transaction statements.

- on return from the subroutine, examines the JPL return variable, ***jpl_retcode***. If it is 0, the subroutine, and therefore the transaction, executed successfully. If it is not zero, the subroutine was aborted by a **JAM** or by the error handler. For either type of error, it executes a rollback.

A sample of such a procedure is shown in the JPL code below. The actual transaction statements are executed in the subroutine whose name is passed to this procedure. This transaction handler may be used with the default error handler or with an installed error handler that returns the abort code (1) for all errors.

```
proc tran_handle
{
    parms subroutine
    vars jpl_retcode
    retvar jpl_retcode
# Call the subroutine.
        jpl :subroutine
# Check the value of jpl_retcode. If it is 0, all statements in
# the subroutine executed successfully and the transaction was
# committed. If it is 1, the error handler aborted the
# subroutine. If it is -1, JAM aborted the subroutine. Execute a
# ROLLBACK for all non-zero return codes.
        if jpl_retcode
        {
            msg emsg "Aborting transaction."
            dbms ROLLBACK
        }
        else
        {
            msg emsg "Transaction succeeded."
        }
        return 0
}

proc update_emp
{
    ....
    dbms COMMIT
    return 0
}
```

To execute the update transaction, the application should execute

```
jpl tran_handle update_emp
```

Once `tran_handle` has set up the return variable, it calls the procedure `update_emp`. Whether `update_emp` is successful or unsuccessful, control is always returned to `tran_handle`.

In the engine-specific *Notes*, there is a list and description of the supported transaction commands with more examples.

# JAM/DB*i*
# Reference Guide

# Chapter 12.
# JAM/DBi *Reference Overview*

This guide has a reference chapter on each of the following:

- JAM/DBi global variables
- DBMS commands
- JAM/DBi library functions
- JAM/DBi utilities

Each reference chapter provides a summary of the topic, and a reference page for each command, function, or utility. The reference pages use following notation:

| | |
|---|---|
| `literal` | This font indicates text that the developer will type verbatim. In particular, it is used for all examples and for the names of JAM library functions, JPL commands, or utilities. |
| SMALL CAPS | Uppercase is used for SQL keywords and dbms command keywords. This use of case is stylistic. Case is significant only for identifiers—names of fields, columns, tables, variables. functions, etc. |
| ***Italics*** | Bold italics show where variable or procedure names should appear. Text in this font should be replaced with a value appropriate for the application. |
| [*x*] | Brackets indicate an optional element. The brackets should not be typed. |
| {*x* \| *x* } | Braces indicate a series of valid options. At least one option must be used. The braces should not be typed. |
| *x*... | Ellipses indicate that an element may be repeated one or more times. |

*Chapter 13.*

# *DBMS Global Variables*

This chapter summarizes and categorizes the JAM/DB*i* global variables.

## 13.1. VARIABLE OVERVIEW

The global JAM/DB*i* variables are automatically defined by JAM/DB*i* at initialization. All JAM/DB*i* global names begin with the characters `@dm`. Since the character `@` is not permitted in user-defined JAM variables, these variables will never conflict with any screen, LDB, or JPL variables defined by your application.

These variables and their values are available to JPL commands and to JAM library functions like `sm_n_getfield` and `sm_n_fptr`.

The variables are automatically maintained by JAM/DB*i*. Before executing a `dbms` or `sql` statement, JAM/DB*i* clears the contents of all its global variables. After executing the statement and before returning control to the application, JAM/DB*i* updates the variables to indicate the current status.

### 13.1.1. Error Data

| | |
|---|---|
| `@dmretcode` | JAM/DB*i* error code. Codes are the same for all engines. |
| `@dmretmsg` | JAM/DB*i* error message. Messages are the same for all engines. |
| `@dmengerrcode` | Engine error code. Codes are unique to the engine. |

| | |
|---|---|
| `@dmengerrmsg` | Engine error message. Messages are unique to the engine. Some engines do not supply messages. |

13.1.2.
## Status Data

| | |
|---|---|
| `@dmretcode` | JAM/DB*i* status code for "no more rows" or "end of proc." |
| `@dmretmsg` | JAM/DB*i* status message for "no more rows" or "end of proc." |
| `@dmengreturn` | Engine return code from a stored procedure. Not used by all engines. |
| `@dmrowcount` | Count of the number of rows fetched to **JAM** by the last SELECT or CONTINUE. Used by all engines. |
| `@dmserial` | A serial value returned after inserting a row into a table with a serial column. Not used by all engines. |
| `@dmengwarncode` | A code or byte signalling a non-fatal error or unusual condition. Used by all engines. |
| `@dmengwarnmsg` | A message corresponding to an engine warning code. Not used by all engines. |

13.2.
# VARIABLE REFERENCE

The rest of this chapter contains a reference page for each global variable. Since some variables store engine-specific values, additional information is provided in the engine-specific *Notes*.

Each reference page has the following sections:

- A description of the variable.
- A list of related variables and commands.
- An example.

The variables are documented in alphabetical order.

# @dmengerrcode

## contains an engine-specific error code

### DESCRIPTION

JAM/DB*i* sets this variable to 0 before executing a `dbms` or `sql` statement. If the engine detects an error, JAM/DB*i* writes the engine's error code to `@dmengerrcode`.

Note that a 0 value in this variable does not guarantee that the last statement executed without error. Some errors are detected by JAM/DB*i* before a request is made to the engine. For example, if an application attempts a SELECT before declaring a connection, JAM/DB*i* detects the error. Use the global variable `@dmretcode` to check for JAM/DB*i* errors.

Because the value of `@dmengerrcode` is engine-specific, the use of an installed error handler is strongly recommended. The application may test for engine-specific errors within the error handler or in a multi-engine application, the error handler may call another function to do this.

Please consult the engine-specific *Notes* for more information about the codes for your engine.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

### RELATED VARIABLES

`@dmengerrmsg`

`@dmretcode`

`@dmretmsg`

### EXAMPLE

```
proc dbi_errhandle
parms stmt engine flag
if @dmengerrcode == 0
   msg emsg @dmretmsg
else if engine == "xyzdb"
   jpl xyzerror @dmengerrcode
else if engine == "oracle"
```

```
   jpl oraerror @dmengerrcode
else
   msg emsg "Unknown engine."
return 1

proc xyzerror
# Check for specific xyzdb error codes.
parms error
   if error == 90931
      msg emsg "Invalid user name."
   else if error == ...
   ...
   else
      msg emsg @dmengerrmsg
   return
```

# @dmengerrmsg

## contains an engine-specific error message

### DESCRIPTION

JAM/DB*i* clears this variable before executing a new `@dbms` or `@sql` statement. If the engine returns an error message after attempting to execute the statement, JAM/DB*i* writes the message to this variable.

If `@dmengerrcode` is 0, this variable contains no message.

Please consult the engine-specific *Notes* for more information about the error messages for your engine.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

### RELATED VARIABLES

`@dmengerrcode`

`@dmretcode`

`@dmretmsg`

### EXAMPLE

```
proc dbi_errhandle
parms stmt engine flag
if @dmengerrcode == 0
   msg emsg @dmretmsg
else
   msg emsg @dmretmsg "%N" @dmengerrmsg
return 1
```

# @dmengreturn

## contains a return code from a stored procedure

### DESCRIPTION

If your engine supports stored procedures and stored procedure return codes, use this variable to get a procedure's return or status code.

By default, JAM/DB*i* will pause the execution of a stored procedure if the procedure executes a SELECT statement and the number of rows in the SELECT set is greater than the number of occurrences in the JAM destination variables. The application must execute DBMS CONTINUE or DBMS NEXT to resume execution. If the value of `@dmengreturn` is null after calling a stored procedure, the procedure may be pending. If the engine has completed the execution of the procedure, `@dmretcode` will contain the `DM_END_OF_PROC` code and `@dmengreturn` will contain the procedure's return code.

Note that the value of this variable will be cleared once another `dbms` or `sql` statement is executed. If the application needs this value for a longer period of time, it should copy it to a standard JAM variable or some other static location.

### SEE ALSO

*Notes*

### RELATED FUNCTIONS

dbms [WITH CURSOR ***cursor***] NEXT

dbms [WITH CURSOR ***cursor***] SET \
[SINGLE_STEP|STOP_AT_FETCH|EXECUTE_ALL]

### RELATED VARIABLES

`@dmretcode`

`@dmretmsg`

### EXAMPLE

```
# create proc checkid @id char(15) as
# declare @idcount int
# select @idcount = SELECT COUNT (*) FROM products WHERE
# id = @id
# if @idcount == 1
#     return 1
```

```
# else
#     return -1


sql  EXEC checkid :+id
if @dmengreturn == 1
   jpl addrow
else if @dmengreturn == -1
   msg emsg "Sorry, " id " is not a valid code."
return
```

# @dmengwarncode

## contains an engine-specific warning code

### DESCRIPTION

Most engines supply a mechanism for signalling an unusual, but non-fatal condition.

Some engines use an eight-element array. If there is a warning, it sets the first element to indicate a warning and then sets one or more additional elements to describe the warning. Other engines uses codes and messages similiar to those it uses for errors. Those of a high severity are handled as errors and those of a low severity are handled as warnings. Please consult the engine-specific *Notes* for information about your engine and for an example.

By default, JAM/DB*i* ignores warnings. If an application needs to alert users to warning codes, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning codes is with an installed exit handler.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

### RELATED VARIABLES

`@dmengwarnmsg`

# @dmengwarnmsg

## contains an engine-specific warning message

### DESCRIPTION

Most engines supply a mechanism for signalling an unusual, but non-fatal condition. Some engines uses a warning array or byte. These engines do not supply warning messages and therefore do not use `@dmengwarnmsg`. Others use a code and message for low-severity errors. Please consult the engine-specific *Notes* for information about your engine and for an example.

By default, JAM/DB*i* ignores warnings. If an application needs to alert users to warning codes or messages, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning values is with an installed exit handler.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

### RELATED VARIABLES

`@dmengwarncode`

# @dmretcode

## contains an engine-independent error or status code

### DESCRIPTION

Before executing a new `dbms` or `sql` statement, JAM/DB*i* writes a 0 to `@dmretcode`. If the statement fails because of a JAM/DB*i* or engine error, JAM/DB*i* writes an error code to `@dmretcode` describing the failure. These codes are defined in `dmerror.h` and are engine-independent. The codes are 5-digits long. See *Appendix* B. for a listing.

Usually a non-zero value in `@dmretcode` indicates that an error occurred. The default or an installed error handler is called for an error. If the default handler is in use, JAM/DB*i* will display an error message. If the application has installed its own error handler, the installed function controls what messages are displayed. Since these codes are generic, applications often need engine-specific error values as well. Engine-specific error codes are written to `@dmengerrcode`.

There are two non-zero codes for `@dmretcode` which are not errors: `DM_NO_MORE_ROWS` and `DM_END_OF_PROC`. When an engine indicates that it has returned all rows for a SELECT set, JAM/DB*i* writes the `DM_NO_MORE_ROWS` code to `@dmretcode`. Since this is not considered an error, JAM/DB*i* does not call the default or an installed error handler. You may test for `DM_MORE_ROWS` after executing a SELECT or in an exit handler. JAM/DB*i* uses `DM_END_OF_PROC` with engines that support stored procedures. When an engine indicates that it has completed executing the stored procedure, JAM/DB*i* writes the `DM_END_OF_PROC` code to `@dmretcode`. This is not an error. An application may test for this code in an exit procedure or after calling a stored procedure. See the engine-specific *Notes* for information on stored procedures.

### SEE ALSO

JAM/DB*i Developer's Guide*, Section 9.2. and Chapter 10..

*Appendix* B.

### RELATED FUNCTIONS

dbms ONERROR [JPL ***entrypoint*** | CALL ***function***]

dbms ONEXIT [JPL ***entrypoint*** | CALL ***function***]

### RELATED VARIABLES

`@dmengerrcode`

`@dmengerrmsg`

```
@dmretmsg
```

**EXAMPLE**

```
# The following are LDB constants.
# DM_ALREADYON = 53251
# DM_LOGON_DENIED = 53253
# DM_NO_MORE_ROWS = 53256


proc dbi_errhandle
parms stmt engine flag
# Check for logon errors.
if @dmretcode == DM_ALREADYON
   return 0
else if @dmretcode == DM_LOGON_DENIED
   msg emsg @dmretmsg "%N" @dmengerrmsg
....
return 1

proc dbi_exithandle
parms stmt engine flag
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg "All rows returned."
return 0
```

# @dmretmsg

## contains an engine-independent error or status message

### DESCRIPTION

Before executing a new `dbms` or `sql` statement, JAM/DB*i* clears `@dmretmsg`. If the statement fails because of a JAM/DB*i* or engine error, JAM/DB*i* writes an error message to `@dmretmsg` describing the failure. These messages are defined in JAM's `msgfile` and are engine-independent. See *Appendix* B. for a listing.

Note that if `@dmretcode` is 0, `@dmretmsg` is always empty.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

`dbms ONEXIT [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

### RELATED VARIABLES

`@dmengerrcode`

`@dmengerrmsg`

`@dmretcode`

### EXAMPLE

```
proc dbi_errhandle
parms stmt engine flag
   msg emsg "Statement " stmt " failed." \
   @dmretmsg "%N" @dmengerrmsg
   return 1
```

# @dmrowcount

## contains a count of the number of rows fetched to JAM by a SELECT or CONTINUE

### DESCRIPTION

Before executing a new `dbms` or `sql` statement, JAM/DB*i* writes a 0 to this variable. If the statement fetches rows, JAM/DB*i* updates the variable writing the number of rows fetched to JAM variables.

Most SQL syntaxes provide an aggregate function COUNT to count the number of values in a column or the number of rows in a SELECT set. The value of `@dmrowcount` is NOT the number of rows in a SELECT set; rather, it is the number of rows returned to JAM variables. Therefore if a SELECT set has 14 rows in total, and its target JAM variables are arrays, each with ten occurrences, `@dmrowcount` will equal 10 after the SELECT is executed, and 4 after the DBMS CONTINUE is executed. If DBMS CONTINUE were executed a second time, `@dmrowcount` would equal 0.

The integer written to `@dmrowcount` is either less than or equal to the maximum number of rows that can be written to the targeted JAM destinations; the maximum number of rows is the number of occurrences in a destination variable. If the value in `@dmrowcount` is less than the maximum number of occurrences, then the entire SELECT set was written to the target variables and no further processing is needed. If `@dmrowcount` equals the maximum number of occurrences, then the SELECT may have fetched more rows than will fit in the variables. To display the rest of the SELECT set, the application must execute DBMS CONTINUE until `@dmrowcount` is less than the maximum number of occurrences (or equals 0) or until `@dmretcode` receives the `DM_NO_MORE_ROWS` code.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

### RELATED VARIABLES

`@dmretcode`

## EXAMPLE

```
proc get_selection
   sql SELECT * FROM movie_archive WHERE subject=:+subj
   jpl check_count
   return

 proc check_count
 # If rows are returned but not the NO_MORE_ROWS code,
 # let the user know there are rows pending.
   if (@dmrowcount > 0) && (@dmretcode != DM_NO_MORE_ROWS)
      msg setbkstat "Press %KPF1 to see more."
   else
      msg setbkstat "All rows returned."
   return

 proc get_more
 # This function is called by pressing PF1.
 # It retrieves the next set of rows.
   dbms CONTINUE
   jpl check_count
   return
```

# @dmserial

## contains a serial column value after performing INSERT

### DESCRIPTION

Some engines supply the datatype `serial` to assist endusers and applications that need to assign a unique numeric value to each row in a table. When an application inserts a row in a table with a serial column, the engine generates a serial number, inserts the row with the number, and returns the number to the application. See the engine-specific *Notes* for information about support for this on your engine.

Before executing a new `dbms` or `sql` statement, **JAM/DB*i*** writes a 0 to `@dmserial`. If the statement is an INSERT and the engine returns a serial value, **JAM/DB*i*** writes the value to `@dmserial`. Since this variable is cleared before executing a new `sql` or `dbms` statement, you must copy its value to another location if you wish to use the value in subsequent statements.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

### EXAMPLE

```
# Column order_num is a SERIAL column.

proc new_order
vars i(3) order_id(5)

    dbms BEGIN
    # First INSERT row into invoices table.
    # Column order_id in table invoices is a SERIAL.
       sql INSERT INTO invoices \
           (order_id, order_date, cust_num) VALUES \
           (0, :+today, :+cust_num)

    # Copy the serial value to a JPL variable for use with
    # subsequent INSERTS.
       cat order_id @dmserial

    # Use order number to insert new rows to the orders
    # table. Column order_id in table orders is an INT.
       for i=1 while i<=max step 1
```

```
        sql INSERT INTO orders \
        (order_id, part_id, quant, u_cost) VALUES \
        (:order_id, :+part_id[i], :+quant[i], :+u_cost[i])
     dbms COMMIT

  msg emsg "Order completed. Invoice number is " order_num
     return
```

# Chapter 14.
# DBMS Commands

This chapter summarizes and categorizes the DBMS commands supported by all engines. These commands are executed with the JPL command dbms or the C library function dm_dbms. Commands that are specific to an engine are described in *Notes*. This includes the transaction commands and any special feature commands.

## 14.1. DBMS COMMAND OVERVIEW

The DBMS commands fall into several categories. The sections below summarize the commands in each category. Some commands may be listed more than once.

### 14.1.1. Engine Selection

| | |
|---|---|
| ENGINE | set the default engine for the application |
| WITH ENGINE | set the default engine for the duration of a command |

### 14.1.2. Using Connections

| | |
|---|---|
| CLOSE CONNECTION | close a named connection |

| | |
|---|---|
| `CLOSE_ALL_CONNECTIONS` | close all connections on all engines |
| `CONNECTION` | set a default connection and engine for the application |
| `DECLARE CONNECTION` | declare a named connection to an engine |
| `WITH CONNECTION` | set the default connection for the duration of a command |

14.1.3.
## Using Cursors

| | |
|---|---|
| `CLOSE CURSOR` | close a cursor |
| `CONTINUE` | fetch the next screenful of rows from a SELECT set |
| `DECLARE CURSOR` | declare a named cursor |
| `EXECUTE` | execute a named cursor |
| `WITH CURSOR` | specify the cursor to use for a statement |

14.1.4.
## Changing SELECT Behavior

| | |
|---|---|
| `ALIAS` | name a **JAM** variable as the destination of a selected column or an aggregate function |
| `BINARY` | create a **JAM**/DB*i* variable for fetching binary values |
| `CATQUERY` | redirect SELECT results to a file or a **JAM** variable |
| `FORMAT` | format the results of a CATQUERY |
| `OCCUR` | set the number of rows for **JAM**/DB*i* to fetch to an array and choose an occurrence where **JAM**/DB*i* should begin writing result rows |
| `START` | set the first row for **JAM**/DB*i* to return from a SELECT set |
| `UNIQUE` | suppress repeating values in a selected column |

14.1.5.
## Paging through Multiple Rows

| | |
|---|---|
| `CONTINUE` | fetch the next screenful of rows from a SELECT set |

| | |
|---|---|
| `CONTINUE_BOTTOM` | fetch the last screenful of rows from a SELECT set |
| `CONTINUE_DOWN` | fetch the next screenful of rows from a SELECT set |
| `CONTINUE_UP` | fetch the previous screenful of rows from a SELECT set |
| `CONTINUE_TOP` | fetch the first screenful of rows from a SELECT set |
| `STORE FILE` | store the rows of a SELECT set in a temporary file so that the application may scroll through the rows |

14.1.6.
## Handling Binary Data

| | |
|---|---|
| `BINARY` | define one or more binary variables |

14.1.7.
## Status and Error Processing

| | |
|---|---|
| `ONENTRY` | install a JPL procedure or C function which JAM/DB*i* will call before executing a `sql` or `dbms` statement |
| `ONERROR` | install a JPL procedure or C function which JAM/DB*i* will call whenever a `sql` or `dbms` statement fails |
| `ONEXIT` | install a JPL procedure or C function which JAM/DB*i* will call after executing a `sql` or `dbms` statement |

14.2.
# COMMAND REFERENCE

The rest of this chapter contains a reference page for each DBMS command. The commands in this chapter may be executed with the JPL command `dbms` or the library function `dm_dbms`. Some engines may support additional commands. See the engine-specific *Notes* for a list of such commands.

Each reference page has the following sections:

- A synopsis of the command, including a listing of available keywords and arguments.

- A description of the command.
- A list of related commands.
- An example.

# ALIAS

## set aliases for a declared or default SELECT cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] ALIAS [ column jamvar \
    [, column jamvar ...] ]

dbms [WITH CURSOR cursor] ALIAS [ jamvar [, jamvar ...] ]
```

### DESCRIPTION

By default, database values are written to JAM variables with the same names as the selected columns. Use this command to map a database column or value to any JAM variable.

If a column name is given, the column is associated with the variable name that follows it. For example,

```
dbms ALIAS lastname emp_lastname, street address
```

If the column `lastname` is selected with the default cursor, JAM/DB*i* will write its values to the JAM variable `emp_lastname`. If the column `street` is selected with the default cursor, JAM/DB*i* will write its values to the JAM variable `address`. For all other columns selected with the default cursor, JAM/DB*i* will write to a variable with the same (unqualified) name as the selected column.

If ***column*** contains characters not permitted in JAM identifiers, enclose ***column*** in quotes to ensure correct parsing.

The case of ***column*** should match the setting of the case flag used to initialize the engine in `dbiinit.c`. For example, if the case flag is `DM_FORCE_TO_LOWER_CASE`, ***column*** must be typed in lower case. The case of ***jamvar*** must be the case used to name the JAM variable. If ***jamvar*** does not exist, JAM/DB*i* ignores the column when it executes the SELECT.

If no ***column*** arguments are given, the association is positional. For example,

```
dbms ALIAS emp_var, , abc
```

If the above statement is executed, then each time values are selected with the default cursor, JAM/DB*i* will write the values of the first and third columns to the JAM variables `emp_var` and `abc` respectively. For all other columns selected with the default cursor, JAM/DB*i* will write to a variable with the same (unqualified) name as the selected column. The order of column names in the SELECT statement determines the mapping. The case of *jamvar* must be the case used to name the **JAM** variable. If *jamvar* does not exist, JAM/DB*i* simply ignores the column when it executes the SELECT.

Named and positional aliases may not be assigned in a single statement.

If aliases are declared for a CATQUERY cursor with the HEADING ON option, JAM/DB*i* will use the aliases rather than the column names to build the heading. The alias for a column selected with a CATQUERY cursor may be enclosed in quotes. This permits a column heading to use embedded spaces. For example,

```
dbms DECLARE emp_cursor CURSOR FOR \
   SELECT first, last, dept FROM emp
dbms WITH CURSOR emp_cursor CATQUERY TO FILE emp_list
dbms WITH CURSOR emp_cursor ALIAS \
   "First Name", "Last Name", Department
dbms WITH CURSOR emp_cursor EXECUTE
```

Aliasing for a cursor is turned off by executing the DBMS ALIAS command with no arguments. Closing a cursor also turns off aliasing. If a cursor is redeclared without being closed, the cursor keeps the aliases. Aliases do not affect INSERT, UPDATE, or DELETE statements.

This command is necessary if the name of a selected column is not a valid JAM variable name, if the application is selecting values from different tables which use the same column name for different values, or if a selection is not a column value, but the value of an aggregate function.

## SEE ALSO

JAM/DB*i* *Developer's Guide*, page 79.

## RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor ] CATQUERY [TO [FILE] destination \
   [SEPARATOR "text"] [HEADING [ON | OFF]] ]
```

```
[WITH CURSOR cursor]
```

## EXAMPLE

```
# Assign aliases by named for a declared cursor.
dbms DECLARE x CURSOR FOR \
   SELECT lname, fname, code FROM directory
dbms WITH CURSOR x ALIAS \
   lname last, fname first, code dept
dbms WITH CURSOR x EXECUTE
dbms WITH CURSOR x ALIAS
```

```
# Set a positional alias for the 2nd and 4th columns.
# Use automatic mapping for the 1st and 3rd columns.
dbms ALIAS , var_x, , var_y
sql SELECT ssn, last, first, address FROM emp
# DBi will write
#     Column ssn to Variable ssn,
#     Column last to Variable var_x,
#     Column first to Variable first, and
#     Column address to Variable var_y.

# Note how the mappings change when the columns are
# listed in another order.
sql SELECT last, first, address, ssn FROM emp
# DBi will write
#     Column last to Variable last,
#     Column first to Variable var_x,
#     Column address to Variable address, and
#     Column ssn to Variable var_y.
```

# BINARY

## define JAM/DB*i* variables for fetching binary values

### SYNOPSIS

```
dbms BINARY variable [, variable ...]
```

### DESCRIPTION

Many engines support a binary datatype for bytes strings and other non-printable data. An application cannot fetch binary values to **JAM** variables (fields, LDB variables, or JPL variables) but it may fetch them to **JAM**/DB*i* variables defined with the command DBMS BINARY.

***variable*** is the name of the binary variable which **JAM**/DB*i* will create. Its definition may include a number of occurrences and/or a length. If a number of occurrences is supplied, it must be enclosed in square brackets. If a variable length is supplied, it must be enclosed in parentheses. If both are supplied, the number of occurrences must be first. Any of the following are permitted:

```
dbi_binvar
dbi_binvar [10] (255)
dbi_binvar [5]
dbi_binvar (8)
```

Any valid **JAM** variable name is a legal **JAM**/DB*i* variable name. The default number of occurrences is 1, and the default length is the maximum, 255. Memory is allocated for the occurrences when they are used (i.e., when a binary column is fetched).

If an application is selecting a binary column, use this command to create a binary variable for the column. The variable may have the same name as the column, or it may be mapped to the column with DBMS ALIAS. Because a binary variable is a target of a SELECT, **JAM**/DB*i* will examine its number of occurrences when determining how many rows to fetch. Therefore, the binary variable should have the same number of occurrences as the other **JAM** target variables. When searching for target variables, **JAM**/DB*i* searches among the binary variables before searching among the **JAM** variables. The developer is responsible for ensuring that the binary variable names do not conflict with **JAM** variable names.

The only legal use of a binary variable in JPL is in the USING clause of a DBMS EXECUTE statement. If no occurrence is given for the variable, the first occurrence is the default.

Once defined, a binary variable is available to the rest of the application. Note that

```
dbms BINARY dbi_binvar[10]
dbms BINARY timestamp[100]
```

is the same as

```
dbms BINARY dbi_binvar[10]  timestamp[100]
```

To delete all binary variables, execute DBMS BINARY with no arguments:

```
dbms BINARY
```

Several JAM/DB*i* library routines are provided for accessing and manipulating the binary variables. These routines are only available in C. They are described in Chapter 15. and listed below.

### RELATED FUNCTIONS

```
dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

### EXAMPLE

```
# "photo" is a binary column
dbms BINARY dbi_binvar
dbms ALIAS photo dbi_binvar
sql SELECT jobcode, site, photo FROM newbuildings \
   WHERE architect = :+lastname


dbms BINARY lastchanged[20]
sql SELECT id, name, description,
```

# CATQUERY

## concatenate a full result row to a JAM variable or a file

### SYNOPSIS

```
dbms [WITH CURSOR cursor] CATQUERY TO jamvar
    [SEPARATOR "text"] [HEADING [ON | OFF]]

dbms [WITH CURSOR cursor] CATQUERY TO FILE file
    [SEPARATOR "text"] [HEADING [ON | OFF]]
```

### DESCRIPTION

The result columns of a SELECT statement are usually mapped to individual variables. Use CATQUERY to map full result rows to a variable's occurrences or to a text file. The options are described below.

| | |
|---|---|
| WITH CURSOR ***cursor*** | Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor. |
| TO ***jamvar*** | Names a JAM variable as the destination. |
| TO FILE ***file*** | Names a text file as the destination. If the file already exists, it is overwritten when the SELECT is executed. |
| SEPARATOR "***text***" | Specifies that JAM/DB*i* should use ***text*** to separate column values in a result row. The default is two blank spaces. |
| HEADING ON | Specifies that JAM/DB*i* should put a heading at the beginning of the SELECT results. This is the default for a catquery to a file. The heading is built using the column names or any aliases assigned to the cursor. The maximum length of a heading is 255 characters. Any additional characters are truncated. |
| HEADING OFF | Specifies that JAM/DB*i* should not use a heading. This is the default for a catquery to a JAM variable. |

JAM/DB*i* attempts to format the column values by searching for JAM variables of the same name and using their attributes for length, precision, and date-time or currency edits. The application may override any default formatting with the command DBMS FORMAT.

The catquery for a cursor is turned off by executing the DBMS CATQUERY command with no arguments. Closing a cursor also turns off the catquery. If a cursor is redeclared without being closed, the cursor keeps the catquery destination as the cursor's SELECT destination.

### Catquery to a Variable

When the catquery destination is a JAM variable, JAM/DB*i* concatenates a result row and writes it to ***jamvar*** when the SELECT is executed. If ***jamvar*** is an LDB or field array, JAM/DB*i* writes the result rows to the array occurrences. If there are more result rows than occurrences in ***jamvar***, use DBMS CONINUE to fetch the additional rows.

If the clause HEADING ON is used, JAM/DB*i* creates a heading by using the cursor's aliases and column names. If ***jamvar*** has two or more occurrences, JAM/DB*i* will put the heading in the first occurrence of ***jamvar***.

### Catquery to a Text File

When the catquery destination is a text file, JAM/DB*i* writes all the result rows to the specified text file when the SELECT is executed. Any existing file with the same name is overwritten. If a result row is longer than the page width, JAM/DB*i* wraps the row to the next line.

If aliases have been specified for the cursor, JAM/DB*i* uses those aliases as column headings in the text file. If there are no aliases, JAM/DB*i* uses the columns' names. If the clause HEADINGOFF is used, JAM/DB*i* does not print a heading.

Since all result rows are written to the file, the DBMS CONTINUE commands should not be used with a CATQUERY TO FILE cursor while the file is open.

The file remains open until DBMS CATQUERY is reset or the cursor is closed.

## RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] ALIAS [column] "text"...

dbms [WITH CURSOR cursor] FORMAT [column] format ...
```

## EXAMPLE

```
# select an employee's first and last name
# and concatenate the values in the field "fullname"
 dbms DECLARE name_cursor CURSOR FOR \
   SELECT last, first WHERE ssn=:+ssn
 dbms WITH CURSOR name_cursor CATQUERY TO fullname \
   SEPARATOR ", "
 dbms WITH CURSOR name_cursor EXECUTE

# select the maximum value from the column "cost"
# and write it to the JPL variable "hi_cost"
# formatting it with currency edit saved with the LDB
# variable "money_var"
 vars hi_cost
 dbms DECLARE max_cursor CURSOR FOR \
```

```
  SELECT MAX(cost) FROM inventory
dbms WITH CURSOR max_cursor CATQUERY TO hi_cost
dbms WITH CURSOR max_cursor FORMAT money_var
dbms WITH CURSOR max_cursor EXECUTE

# Write the results of the default SELECT cursor
# to a file with heading. Turn off ALIAS and CATQUERY
# when finished.
dbms CATQUERY TO FILE phonelist
dbms ALIAS emplast "Last Name", empfirst "First Name",\
  phone1 "Main Number", phone2 "Additional Number"
sql SELECT emplast, empfirst, phone1, phone2 FROM emp
dbms CATQUERY
dbms ALIAS
```

# CLOSE_ALL_CONNECTIONS

## close all connections on an engine

### SYNOPSIS

```
dbms CLOSE_ALL_CONNECTIONS
```

### DESCRIPTION

When this command is executed, JAM/DB*i* closes every connection which the application declared on any and all engines. For each connection, it closes all cursors belonging to the connection, disconnects from the engine, and frees all structures associated with the connection.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, page 55.

### VARIANTS

dbms CLOSE CONNECTION [***connection***]

### RELATED FUNCTIONS

dbms [WITH ENGINE ***engine***] DECLARE ***connection*** CONNECTION \
FOR [***OPTION arg*** ...]

# CLOSE CONNECTION

## close a declared connection

### SYNOPSIS

```
dbms CLOSE CONNECTION [connection]
```

### DESCRIPTION

Executing this command closes all open cursors associated with the named or default connection, logs off the connection from its engine, and frees the connection data structure.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, 55.

### VARIANTS

```
dbms [WITH ENGINE engine] CLOSE_ALL_CONNECTIONS
```

### RELATED FUNCTIONS

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
   FOR [OPTION arg ...]
```

```
WITH CONNECTION connection
```

# CLOSE CURSOR

## close a named or default cursor

### SYNOPSIS

```
dbms CLOSE CURSOR [cursor]

dbms WITH CONNECTION connection CLOSE CURSOR
```

### DESCRIPTION

Use this command to close an open cursor. Closing a cursor frees all structures associated with the cursor.

Closing a cursor is convenient way of turning off all attributes assigned to the cursor with DBMS ALIAS, DBMS CATQUERY, DBMS FORMAT, DBMS OCCUR, DBMS START, DBMS STORE_FILE, DBMS TYPE, and DBMS UNIQUE.

If ***cursor*** is not given, JAM/DB*i* closes the default SELECT cursor. A connection may also be specified when closing a default cursor. JAM/DB*i* will automatically declare another default SELECT cursor when needed. A connection name should not be given when closing a named cursor.

Closing a connection also closes all cursors associated with the connection.

### SEE ALSO

JAM/DB*i Developer's Guide*, page 57.

### VARIANTS

```
dbms [WITH ENGINE engine] CLOSE CONNECTION [connection]

dbms CLOSE_ALL_CONNECTIONS
```

### RELATED FUNCTIONS

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQLstmt

dbms WITH CURSOR cursor EXECUTE

WITH CURSOR cursor
```

## EXAMPLE

```
# Assign a catquery and aliases to the default SELECT
 # cursor. Close the cursor when finished.
 dbms CATQUERY TO FILE phonelist
 dbms ALIAS emplast "Last Name", empfirst "First Name",\
   phone1 "Main Number", phone2 "Additional Number"
 sql SELECT emplast, empfirst, phone1, phone2 FROM emp
 dbms CLOSE CURSOR
```

# CONNECTION

## set or change the default connection

### SYNOPSIS

dbms CONNECTION ***connection***

### DESCRIPTION

If an application has declared two or more connections, the application may set a default connection with this command. The default connection is used for all subsequents statements that do not use a WITH CONNECTION or WITH CURSOR clause.

### RELATED FUNCTIONS

dbms CLOSE CONNECTION ***connection***

dbms [WITH ENGINE ***engine***] DECLARE CONNECTION ***connection***

WITH CONNECTION ***connection***

### EXAMPLE

```
dbms ENGINE sybase
dbms DECLARE a CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER s1 DATABASE master
dbms DECLARE b CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER s2 DATABASE projects
dbms CONNECTION a
dbms WITH CONNECTION b DECLARE c1 CURSOR FOR \
   INSERT finance (number, title, manager) \
   VALUES (::number, ::title, ::manager)
```

# CONTINUE

## fetch the next set of rows associated with a default or named SELECT cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE
```

### DESCRIPTION

If a SELECT retrieves more rows than will fit in its destination variables, JAM/DB*i* will return only as many rows as will fit. It continues fetching more rows from the SELECT set when the application executes this command. If there are pending rows, executing this command clears the destination variables, and fetches the next screenful of rows from the SELECT set. If there are no pending rows, executing this command does nothing.

DBMS CONTINUE is always associated with a SELECT cursor. If no cursor is named, JAM/DB*i* uses the default SELECT cursor.

Note that if the cursor's aliases have changed between the execution of the SELECT and the execution of DBMS CONTINUE, DBMS CONTINUE uses the new settings.

This command should not be used with a CATQUERY TO FILE cursor. CATQUERY TO FILE always writes out the entire select set to the CATQUERY file.

### VARIANTS

```
dbms [WITH CURSOR cursor] CONTINUE_DOWN
```

### RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
dbms [WITH CURSOR cursor] STORE [FILE [file]]
```

### EXAMPLE

```
dbms DECLARE movie_list CURSOR FOR \
   SELECT * FROM movie_archive WHERE subject=::subj

proc get_selection
 dbms WITH CURSOR movie_list EXECUTE USING subject
```

```
jpl check_count
return


proc check_count
# DM_NO_MORE_ROWS is an LDB constant equal to 53256
if @dmretcode != DM_NO_MORE_ROWS
  msg setbkstat "Press %KPF1 to see more films " \
     "or press %KPF2 to specify another topic."
else
  msg setbkstat "That's all folks!"
return

proc get_more
# This function is called by pressing PF1.
# It retrieves the next set of rows.
dbms WITH CURSOR movie_list CONTINUE
jpl check_count
return
```

# CONTINUE_BOTTOM

## fetch the last page of rows associated with the default or named SELECT cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
```

### DESCRIPTION

This command fetches the last screenful of rows from the cursor's SELECT set. If no cursor is named, JAM/DB*i* uses the default SELECT set. If number of rows in the SELECT set is less than the number of occurrences in the JAM variables, JAM/DB*i* will ignore the request.

Some engines automatically support this command. Other engines require a temporary storage file created by the command DBMS STORE FILE. If JAM/DB*i* returns DM_BAD_CMD error when the application executes this command, the engine needs a scrolling file. See the engine-specific *Notes* for more information.

This command should not be used with a CATQUERY TO FILE cursor.

### RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE
dbms [WITH CURSOR cursor] CONTINUE_DOWN
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

### EXAMPLE

```
#Engines not requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
dbms WITH CURSOR emp_cursor CONTINUE_BOTTOM

#Engines requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
....
dbms WITH CURSOR emp_cursor CONTINUE_BOTTOM
```

# CONTINUE _DOWN

## fetch the next set of rows associated with the default or named SELECT cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_DOWN
```

### DESCRIPTION

This command is identical to DBMS CONTINUE.

Note that CONTINUE is always associated with a SELECT cursor. If no cursor is named, JAM/DB*i* uses the default SELECT cursor.

### VARIANTS

```
dbms [WITH CURSOR cursor] CONTINUE
```

### RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

### EXAMPLE

```
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
proc get_more
dbms WITH CURSOR emp_cursor CONTINUE_DOWN
```

# CONTINUE_TOP

## fetch the first page of rows associated with the default or named SELECT cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_TOP
```

### DESCRIPTION

This command fetches the first screenful of rows from the cursor's SELECT set. If no cursor is named, JAM/DB*i* uses the default SELECT cursor. If number of rows in the SELECT set is less than the number of occurrences in the **JAM** variables, JAM/DB*i* will ignore the request.

Some engines automatically support this command. Other engines require a temporary storage file created by the command DBMS STORE FILE. If the engine needs such a file and the application tries to execute DBMS CONTINUE_TOP without executing DBMS STORE, JAM/DB*i* returns the error DM_BAD_CMD. See the engine-specific *Notes* for more information.

### RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE

dbms [WITH CURSOR cursor] CONTINUE_BOTTOM

dbms [WITH CURSOR cursor] CONTINUE_DOWN

dbms [WITH CURSOR cursor] CONTINUE_UP

dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

### EXAMPLE

```
#Engine not requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_to_start
dbms WITH CURSOR emp_cursor CONTINUE_TOP
```

```
#Engines requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_to_start
dbms WITH CURSOR emp_cursor CONTINUE_TOP
```

# CONTINUE_UP

## fetch the previous page of rows associated with the default or named SELECT cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_UP
```

### DESCRIPTION

Use this command to scroll backwards through a SELECT set. If no cursor is named, **JAM/**DB*i* uses the default SELECT set. If number of rows in the SELECT set is less than the number of occurrences in the **JAM** variables, JAM/DB*i* will ignore the request.

Some engines automatically support this command. Other engines require a temporary storage file created by the command DBMS STORE FILE. If the engine needs such a file and the application tries to execute DBMS CONTINUE_UP before executing DBMS STORE, JAM/DB*i* returns the error DM_BAD_CMD. See the engine-specific *Notes* for more information.

This command should not be used with a CATQUERY TO FILE cursor.

### RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE

dbms [WITH CURSOR cursor] CONTINUE_BOTTOM

dbms [WITH CURSOR cursor] CONTINUE_DOWN

dbms [WITH CURSOR cursor] CONTINUE_TOP

dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

### EXAMPLE

```
#Engine not requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_back
dbms WITH CURSOR emp_cursor CONTINUE_UP
```

```
#Engines requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_back
dbms WITH CURSOR emp_cursor CONTINUE_UP
```

# DECLARE CONNECTION

## create a named connection to a server and database

### SYNOPSIS

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
    [FOR OPTION arg ...]
```

### DESCRIPTION

Applications which must connect to two or more servers should use this command to declare a named connection to a server. If JAM/DB*i* executes this statement successfully, it allocates a connection structure and adds it to the list of open structures.

The combination of necessary or supported options is engine-specific. See the engine-specific *Notes* in this document for a listing.

The connection remains open until it is closed with `DBMS CLOSE CONNECTION` or `DBMS CLOSE_ALL_CONNECTIONS`.

### RELATED FUNCTIONS

```
dbms CLOSE CONNECTION [connection]

dbms CLOSE_ALL_CONNECTIONS

dbms CONNECTION connection

WITH CONNECTION connection
```

# DECLARE CURSOR

## declare a named cursor for a SQL statement

### SYNOPSIS

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQLstmt
```

### DESCRIPTION

Use this command to create or redeclare a named cursor.

If the application has not already declared ***cursor***, JAM/DB*i* allocates a new cursor structure and adds its name to the list of declared cursors.

If a structure already exists for ***cursor*** and the connection is the same, JAM/DB*i* reinitializes the structure. Reinitialization clears any information on SELECT columns, binding parameters, and the maximum number of rows to fetch. It does not clear any attributes assigned to the cursor with the statements DBMS ALIAS, DBMS CATQUERY, DBMS FORMAT, DBMS OCCUR, DBMS START, DBMS STORE, DBMS TYPE, or DBMS UNIQUE. JAM/DB*i* will use these settings if the cursor is redeclared with a SELECT statement. It will ignore the attributes if the cursor is redeclared with an INSERT, UPDATE, or DELETE statement. To redeclare the cursor with a new (empty) structure, close the cursor with DBMS CLOSE CURSOR before executing the new declaration.

If a cursor is redeclared on another connection, JAM/DB*i* automatically closes the cursor and declares a new structure.

The cursor remains open until it is explicitly closed with the command DBMS CLOSE CURSOR. Closing a connection also closes all cursors on the connection.

There are few restrictions on valid cursor names. However, you should avoid using any SQL or JAM/DB*i* keyword as a cursor name. Please note that JAM/DB*i* is case sensitive regarding cursor names; for example, it interprets cursor `c1` as distinct from cursor `C1`.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, pages 57, 72.

### RELATED FUNCTIONS

```
dbms WITH CURSOR cursor EXECUTE
dbms CLOSE CURSOR cursor
WITH CURSOR cursor
```

**EXAMPLE**

```
dbms WITH ENGINE oracle DECLARE emp_cursor FOR \
   SELECT ss, last, first FROM emp \
   WHERE dept = ::parameter
...
dbms WITH CURSOR emp_cursor EXECUTE USING dept_name
```

# ENGINE

## set or change the default engine

### SYNOPSIS

`dbms ENGINE` ***engine***

### DESCRIPTION

If an application has initialized two or more engines, the application may use this command to set a default engine. If an application has only one initialized engine, JAM/DB*i* automatically assigns that engine as the default.

***engine*** is a mnemonic associated with one of the support routines initialized in `dbiinit.c` or in a call to `dm_init`.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, page 52.

### RELATED FUNCTIONS

`WITH ENGINE` ***engine***

# EXECUTE

## execute the SQL statement declared for a named cursor

### SYNOPSIS

dbms WITH CURSOR ***cursor*** EXECUTE [USING ***args***]

### DESCRIPTION

Use this statement to execute the statement associated with a declared cursor.

This statement does not support the WITH CONNECTION clause. JAM/DB*i* uses the engine that was specified either by name or by default when the cursor was declared. The only way to change the cursor's engine or connection is to redeclare the cursor.

If an application is executing a similar statement many times, it is often more efficient to declare a cursor for the statement. Usually the engine saves the parsed statement, executing it when the application executes the cursor. It is not necessary to redeclare the cursor to supply new data for a WHERE or VALUES clause. Instead, the application may declare the cursor and use a substitution parameter for each value that the application will supply when it executes the cursor. Substitution parameters begin with a double colon (::). For example,

```
dbms DECLARE c1 CURSOR FOR \
   SELECT * FROM titles WHERE author LIKE ::author_parm
```

author_parm is simply a place holder for the value that will be supplied when the cursor is executed. For example,

```
dbms WITH CURSOR c1 EXECUTE USING "Fau%"
```

would fetch rows where author began with the characters "Fau." The application could execute the cursor repeatedly, each time with a new value. It may use the value of a field to supply a value. For example,

```
dbms WITH CURSOR c1 EXECUTE USING aname
```

Since aname is not quoted, JAM/DB*i* assumes it is a JAM variable. If an argument in the USING clause is a field or LDB variable with a date-time, currency, null field, or type edit JAM/DB*i* formats the variable's value before passing it to the engine.

This topic is covered in detail in the *Developer's Guide.*

### SEE ALSO

JAM/DB*i* *Developer's Guide*, page 72.

### RELATED FUNCTIONS

dbms DECLARE ***cursor*** CURSOR FOR ***SQLstmt***

```
dbms CLOSE CURSOR cursor

dbms [WITH CURSOR cursor] CONTINUE

WITH CURSOR cursor
```

## EXAMPLE

```
  dbms DECLARE x CURSOR FOR \
    SELECT * FROM inventory WHERE lname=::p1 OR ss=::p2

# bind by position:
 dbms WITH CURSOR x EXECUTE USING newname, ss_number

 # or bind by name:
 dbms WITH CURSOR x EXECUTE \
   USING p1 = newname, p2 = ss_number
```

# FORMAT

## format CATQUERY values

### SYNOPSIS

```
dbms [WITH CURSOR cursor] FORMAT \
    [[column] format [, [column] format ...]]
```

### DESCRIPTION

Use this command to format CATQUERY values before writing them to a variable or a text file. The options are explained below.

| | |
|---|---|
| WITH CURSOR ***cursor*** | Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor. |
| ***column*** | Names a selected column. The case of ***column*** should match the setting of the case flag for the engine in `dbiinit.c`. If columns are not named, the formats are applied by position. |
| ***format*** | Describes how JAM/DB*i* should format the value. ***format*** is either a JAM variable or a quoted precision edit. |

If ***format*** is a JAM variable, JAM/DB*i* formats the column value as if it were writing to the field. In particular, the following characteristics will affect the formatting:

- variable's maximum shifting length
- variable's JAM type

See Section 9.1.3. in the *Developer's Guide* of this document for more information about formatting with JAM type.

***format*** may also be a precision edit. A precision edit is a quoted string beginning with a percent sign. It supplies the length of the value, and optionally, a decimal precision for numeric values.

A precision is given in the form

***"%width"***

***"%width.precision"***

To turn off formatting on the default or named cursor, execute the command with no arguments.

## EXAMPLE

```
# use column "lastname" exactly as returned
# format column "revdate" with the LDB variable "today",
# format column "sal" to width 15 with 2 decimal places,
# format column "comment" to width 30 and truncate excess
dbms CATQUERY TO FILE listing
dbms FORMAT revdate today, sal "%15.2", comment  "%30"
sql SELECT lastname, sal, revdate, comment FROM employee
```

# OCCUR

## change the behavior of a SELECT cursor that writes to JAM arrays

### SYNOPSIS

```
dbms [WITH CURSOR cursor] OCCUR  occ_int [MAX int]
dbms [WITH CURSOR cursor] OCCUR CURRENT [MAX int]
```

### DESCRIPTION

By default, if the destination of a SELECT is one or more arrays, JAM/DB*i* fetches as many rows as will fit in the arrays and begins writing at the first occurrence in the arrays. Use this command to change the default behavior for a SELECT cursor. The options for the command are:

| | |
|---|---|
| WITH CURSOR ***cursor*** | Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor. |
| ***occ_int*** | Specifies the occurrence number where JAM/DB*i* should begin placing SELECT results. |
| CURRENT | Specifies that JAM/DB*i* should use the occurrence number of the "current" field. JAM/DB*i* begins writing at this occurrence number in the target arrays. Note that the current field is the one containing the JAM screen cursor and is not necessarily a target variable. |
| MAX ***int*** | Specifies the maximum number of rows to fetch for a SELECT or CONTINUE. If ***int*** is less than 1, no rows are fetched. |

The setting is turned off by executing the DBMS OCCUR command with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for SELECT's and CONTINUE's.

DBMS OCCUR is ignored with a CATQUERY cursor.

### RELATED FUNCTIONS

```
[WITH CURSOR cursor]
```

## EXAMPLE

```
dbms DECLARE title_cursor CURSOR FOR \
   SELECT * FROM booklist WHERE isbn = :+code
 dbms WITH CURSOR title_cursor OCCUR CURRENT
 dbms WITH CURSOR title_cursor EXECUTE
```

# ONENTRY

## install an entry function

### SYNOPSIS

```
dbms ONERROR CALL function
dbms ONERROR JPL jpl_entry_point
```

### DESCRIPTION

Use this command to install a JPL routine or a C function which JAM/DB*i* will call before it executes a `sql` or `dbms` statement.

Currently, this function is for informational purposes only. For instance, you may wish to log statements to a file on disk before executing them. You may use this function with an exit handler to track the start and complete time for a query or any database other operation.

The function is passed three arguments:

1. a copy of the first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word `sql` or `dbms`
2. the name of the engine where
3. context flag; for the entry handler its value is 1.

The function's return code is not used.

If the error occurred while executing a JPL statement with the command `dbms` or `sql`:

- 0 returns control to the JPL procedure where the error occurred
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either JAM or another JPL procedure)

If the error occurred while executing a statement with one of the `dm_` library functions, the `dm_` function returns the error handler's return code.

To use a C function as an error handler, you must first install the function as a prototyped function. Please consult the JAM *Programmer's Guide* for more information.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, page 93.

JAM/DB*i* *Reference Guide*, global variables, page 113

## RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

# ONERROR

## set the behavior of the error handler

### SYNOPSIS

```
dbms ONERROR CALL function
dbms ONERROR CONTINUE
dbms ONERROR JPL jpl_entry_point
dbms ONERROR STOP
```

### DESCRIPTION

Use this command to set or change the behavior of the JAM/DB*i* error handler for the application. The default error handler displays an error message. The source of the message is determined by the engine's initialization. If an engine is initialized with the flag `DM_DEFAULT_ENG_MSG` the default error handler displays an engine-specific error message. If it is initialized with the flag `DM_DEFAULT_DBI_MSG` the error handler uses messages only from the JAM message file. If an error occurs while executing a JPL procedure, the default handler aborts the procedure, returning –1 to the calling procedure.

An application may override the default error handler with the command DBMS ONERROR and an argument. Please note that the error handler is global to the application. Each execution of this command overrides the previous error handler.

The command variants are explained below.

#### ONERROR STOP

This command restores the default error handler.

#### ONERROR CONTINUE

This command prevents the default error handler from aborting a JPL procedure where a JAM/DB*i* error occurs. Message display is not changed.

#### ONERROR JPL or ONERROR CALL

These commands install a user function as the error handler. If JAM/DB*i* or the engine find an error, JAM/DB*i* updates the global error and status variables (i.e., `@dm` variables) and calls the installed function.

The function displays any error messages and its return code controls whether or not JPL execution is aborted.

The function is passed three arguments:

1. the first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word `sql` or `dbms`

2. the name of the engine for the attempted statement
3. context flag; for the error handler its value is 2.

The function's return code is returned to the application.

If the error occurred while executing a JPL statement with the command `dbms` or `sql`:

- 0 returns control to the JPL procedure where the error occurred
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either JAM or another JPL procedure)

If the error occurred while executing a statement with one of the `dm_` library functions, the `dm_` function returns the error handler's return code.

To use a C function as an error handler, you must first install the function as a prototyped function. Please consult the JAM *Programmer's Guide* for more information.

## SEE ALSO

JAM/DB*i* *Developer's Guide*, page 93.

JAM/DB*i* *Reference Guide*, global variables, page 113

## RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

# ONEXIT

## install an exit handler

### SYNOPSIS

```
dbms ONEXIT CALL function
dbms ONERROR JPL jpl_entry_point
```

### DESCRIPTION

Use this command to install a function which **JAM/DB*i*** will call after executing a `dbms` or `sql` command from JPL or C. You may use this function to process error and status codes after every command.

Installing an `ONEXIT` function will override the default error handler. Please note that the exit handler is global to the application. Each execution of this command overrides the previous exit handler.

The function is passed three arguments:

1. the first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word `sql` or `dbms`
2. the name of the engine for the attempted statement
3. context flag; for the exit handler its value is 1.

The function's return code is returned to the application. If an error occurred while executing a JPL statement with the command `dbms` or `sql` and there is no `ONEXIT` function, then

- 0 returns control to the JPL procedure where the error occurred
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either **JAM** or another JPL procedure)

If an error occurred while executing a statement with one of the `dm_` library functions and there is no `ONEXIT` function, the `dm_` function returns the exit handler's return code.

To use a C function as an exit handler, you must first install the function as a prototyped function. Please consult the **JAM** *Programmer's Guide* for more information.

### SEE ALSO

**JAM/DB*i*** *Developer's Guide*, page 93.

**JAM/DB*i*** *Reference Guide*, global variables, page 113

## RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

# START

## specify a starting row in a SELECT set

### SYNOPSIS

dbms [WITH CURSOR ***cursor***] START [***int***]

### DESCRIPTION

By default, when a SELECT set contains more than one row, JAM/DB*i* fetches them sequentially beginning with the first row in the SELECT set. Use this command to begin fetching at row ***int***. JAM/DB*i* will read and discard ***int*** – 1 rows from the SELECT set before returning the requested rows to the application. If the application is counting the rows fetched, the discarded rows do not update @dmrowcount. If ***int*** is greater than the number of rows in the SELECT set, no rows are displayed.

If no cursor is specified, JAM/DB*i* uses the default SELECT cursor.

The setting is turned off by executing DBMS START with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for SELECT's.

### RELATED FUNCTIONS

WITH CURSOR ***cursor***

### EXAMPLE

```
proc discard_100
# dbi_count is an LDB variable
dbms COUNT dbi_count
dbms START 100
sql SELECT * FROM emp
if @dmrowcount == 0
   msg emsg "There are less than 100 employees."
dbms START
return
```

# STORE

## set up a continuation file for a named or default cursor

### SYNOPSIS

```
dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

### DESCRIPTION

When this command is used with a SELECT cursor, JAM/DB*i* maintains a copy of the result rows in a temporary binary file. The use of a file permits an application to scroll forward and backward in a SELECT set, even if the database has no native support for backward scrolling.

If ***filename*** is not given, JAM/DB*i* calls the standard C library routine `tmpfile` to create and open a temporary binary file.

A continuation file remains open for the life of the cursor, or until the feature is turned off with the command,

```
dbms [WITH CURSOR cursor] STORE
```

Executing the command without the keyword FILE closes and deletes the file and turns off the feature for the named or default cursor. Closing the cursor also closes and deletes the file. If a cursor is not closed but simply redeclared for another SELECT statement, the file is cleared. Therefore, a continuation file holds the results of one SELECT statement only.

The use of a continuation file does not force the engine to return the entire SELECT set when the SELECT is executed. In its usual manner, JAM/DB*i* examines the number of occurrences in the destination variable to determine the number of rows to fetch. Each time it fetches rows from the engine by executing the SELECT or a DBMS CONTINUE, JAM/DB*i* updates the screen and appends the new data to the continuation file. If the application wishes to see rows already fetched, JAM/DB*i* uses the continuation file to get the rows and update the screen. If JAM/DB*i* reaches the end of the continuation file and the application executes another DBMS CONTINUE, JAM/DB*i* will attempt to get more rows from the engine. When the engine returns the no-more-rows code, JAM/DB*i* sets `@dmretcode` to the value of `DM_NO_MORE_ROWS`. Similarly, if the application attempts to scroll back past the first row in the continuation file, JAM/DB*i* sets `@dmretcode` to `DM_NO_MORE_ROWS`. See Appendix B. for a list of error and status codes. Write errors are not reported.

This command provides several advantages:

- a means for displaying very large SELECT sets without keeping all rows in memory at once

- better response time for very large SELECT sets; since fetches are incremental it is not necessary to get the entire SELECT set at once

- a means for forcing an engine to release a shared lock on a large SELECT set

Consult the *Notes* for information on engine-specific scrolling issues.

## RELATED FUNCTIONS

dbms [WITH CURSOR ***cursor***] CONTINUE_BOTTOM

dbms [WITH CURSOR ***cursor***] CONTINUE_TOP

dbms [WITH CURSOR ***cursor***] CONTINUE_UP

## EXAMPLE

```
dbms DECLARE emp_cursor CURSOR FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
jpl mapkeys

proc mapkeys
vars SPGU(6) SPGD(6) APP1(6) APP2(6) XLATE(1)
cat SPGU "0x113"
cat SPGD "0x114"
cat APP1 "0x6102"
cat APP2 "0x6202"
cat XLATE "2"
# Set the control strings for APP1 and APP2 on
# this screen to call DBi scroll functions
call sm_putjctrl :APP1 "^jpl scroll_forward" 0
call sm_putjctrl :APP2 "^jpl scroll_back" 0
# Remap the logical page up and down keys to
# APP1 and APP2. (This should be reset on screen exit.)
call sm_keyoption :SPGU :XLATE :APP1
call sm_keyoption :SPGD :XLATE :APP2
return

proc scroll_forward
# SPGU -> APP1 = ^jpl scroll_forward
dbms WITH CURSOR emp_cursor CONTINUE
return
```

```
proc scroll_back
# SPGD -> APP2 = ^jpl scroll_back
dbms WITH CURSOR emp_cursor CONTINUE_UP
return
```

# UNIQUE

## suppress repeating values in selected columns

### SYNOPSIS

```
dbms [WITH CURSOR cursor] UNIQUE column [, column...]
```

### DESCRIPTION

The following command suppresses repeating values in each named column of a SELECT set when the values are in adjacent rows. Typically, this feature is set for a column named in an ORDER BY clause.

The options are

| | |
|---|---|
| WITH CURSOR ***cursor*** | Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor. |
| ***column*** | Specifies a column name in the SELECT statement. |

If no cursor is specified, JAM/DB*i* uses the default SELECT cursor.

If the destination variable has a null edit, an occurrence containing a suppressed value is blank, not null.

The setting is turned off by executing the DBMS UNIQUE command with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for SELECT's and CONTINUE's.

### RELATED FUNCTIONS

WITH CURSOR ***cursor***

### EXAMPLE

```
#Since several items may be ordered on the same invoice,
#suppress repeating invoice numbers when listing
#outstanding sales orders.

dbms DECLARE order_cursor CURSOR FOR \
   SELECT invoice_no, id, desc, quan, cost FROM newsales \
   ORDER BY invoice_no
dbms WITH CURSOR order_cursor UNIQUE invoice_no
dbms WITH CURSOR order_cursor EXECUTE
```

# WITH CONNECTION

## use a named connection for the duration of a statement

### SYNOPSIS

dbms WITH CONNECTION ***connection DBMS_statement***...

sql WITH CONNECTION ***connection SQL_statement*** ...

### DESCRIPTION

This clause specifies a connection for the execution of the command, overriding the default connection. ***connection*** must be declared and open.

Any sql statement may use this clause.

Some dbms statements may also use it. In particular,

dbms [WITH CONNECTION ***connection***] DECLARE ***cursor*** CURSOR...

Once a cursor is declared it remains associated with the connection on which it was declared. After declaring the cursor, the WITH CONNECTION clause should not be used in statements that manipulate the cursor. However, the WITH CONNECTION clause may be used on statements that manipulate the default cursor on any declared connection. Therefore, the following statements:

```
dbms WITH CONNECTION connection ALIAS ...
dbms WITH CONNECTION connection CATQUERY ...
dbms WITH CONNECTION connection CLOSE CURSOR
dbms WITH CONNECTION connection CONTINUE
dbms WITH CONNECTION connection CONTINUE_BOTTOM
dbms WITH CONNECTION connection CONTINUE_TOP
dbms WITH CONNECTION connection CONTINUE_UP
dbms WITH CONNECTION connection FORMAT ...
dbms WITH CONNECTION connection OCCUR ...
dbms WITH CONNECTION connection START ...
dbms WITH CONNECTION connection STORE ...
dbms WITH CONNECTION connection UNIQUE ...
```

perform the request on the default SELECT cursor on the named connection.

Some engine-specific dbms commands may also support the WITH CONNECTION clause. See the engine-specific *Notes* for more information.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, page 55.

Engine-specific *Notes*.

## RELATED FUNCTIONS

dbms [WITH ENGINE ***engine***] DECLARE ***connection*** CONNECTION \
SERVER ***server*** [DB ***database***]

dbms CONNECTION ***connection***

dbms CLOSE CONNECTION [***connection***]

dbms CLOSE_ALL_CONNECTIONS

WITH CURSOR ***cursor***

WITH ENGINE ***engine***

# WITH CURSOR

## use a named cursor for the duration of a statement

### SYNOPSIS

dbms WITH CURSOR ***cursor DBMS_statement***

### DESCRIPTION

This clause specifies the name of a declared cursor on which JAM/DB*i* will execute the dbms command.

Once a cursor has been declared, the application may manipulate or execute the cursor by using the WITH CURSOR clause.

```
dbms WITH CURSOR cursor ALIAS ...
dbms WITH CURSOR cursor CATQUERY ...
dbms WITH CURSOR cursor CONTINUE
dbms WITH CURSOR cursor CONTINUE_BOTTOM
dbms WITH CURSOR cursor CONTINUE_TOP
dbms WITH CURSOR cursor CONTINUE_UP
dbms WITH CURSOR cursor EXECUTE ...
dbms WITH CURSOR cursor FORMAT ...
dbms WITH CURSOR cursor OCCUR ...
dbms WITH CURSOR cursor START ...
dbms WITH CURSOR cursor STORE ...
dbms WITH CURSOR cursor UNIQUE ...
```

If the WITH CURSOR clause is not used with these statements, JAM/DB*i* uses the default SELECT cursor. The application may also manipulate the default cursor by using the WITH CONNECTION clause.

Some engine-specific dbms commands may also support the WITH CONNECTION clause. See the engine-specific *Notes* for more information.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, page 57.

Engine-specific *Notes*.

### RELATED FUNCTIONS

dbms DECLARE ***cursor*** CURSOR FOR ***SQLstmt***

```
dbms CLOSE CURSOR cursor
WITH CONNNECTION connection
WITH ENGINE engine
```

# WITH ENGINE

## use a named engine for the duration of a statement

### SYNOPSIS

`dbms WITH ENGINE` ***engine DBi_command...***

### DESCRIPTION

This clause specifies which engine JAM/DB*i* should use when executing a command. ***engine*** must be an initialized engine. An engine is initialized by using the `vendor_list` structure in `dbiinit.c` or by a call to `dm_init`.

***engine*** must be one of the mnemonics associated with an initialized support routine.

The following commands accept an optional `WITH ENGINE` clause:

dbms WITH ENGINE ***engine*** DECLARE ***connection*** CONNECTION ...

If the `WITH ENGINE` clause is not used, JAM/DB*i* uses the default engine. If only one engine is initialized, that engine is automatically the default. An application using two or more engines may set the default engine with the `DBMS ENGINE` command.

Once a connection is declared it remains associated with the engine on which it was declared. After declaring the connection, the `WITH ENGINE` clause is no longer necessary or valid in any statement except `DBMS CLOSE CONNECTION` if the application wishes to close the default connection on an engine.

### SEE ALSO

JAM/DB*i* *Developer's Guide*, 52.

### RELATED FUNCTIONS

`dbms ENGINE` ***engine***

`WITH CONNECTION` ***connection***

`WITH CURSOR` ***cursor***

# Chapter 15.
# JAM/DB*i* Library Reference

This chapter contains a reference page for each of the JAM library functions.

The library includes functions for initializing JAM/DB*i*, and installing and de-installing an engine at runtime. The functions are:

- `dm_dbi_init`: initialize JAM for use with JAM/DB*i*.
- `dm_init`: initialize an engine.
- `dm_reset`: close all structures associated with an engine.

It includes functions for executing SQL and DBMS commands. The functions are

- `dm_dbms`: execute any DBMS command directly from C.
- `dm_sql`: execute any SQL statement directly from C.
- `dm_dbms_noexp`: like `dm_dbms` except no colon preprocessing is performed.
- `dm_sql_noexp`: like `dm_sql` except no colon preprocessing is performed.

It provides a function for simulating colon-plus processing from C. It is

- `dm_expand`

It provides a function for getting the full text of the last executed `dbms` or `sql` command. It is

- `dm_getdbitext`

The library also provides functions for handling binary values. They are

- `dm_bin_create_occur`
- `dm_bin_delete_occur`

- `dm_bin_get_dlength`
- `dm_bin_get_occur`
- `dm_bin_length`
- `dm_bin_max_occur`
- `dm_bin_set_dlength`

Developers may use these functions in any C hook function. Each reference page has the following sections:

- A synopsis of the function, including a listing of available keywords and arguments.
- A description of the function.
- A list of related functions.
- An example.

# dm_bin_create_occur

## get or allocate an occurrence in a binary variable

### SYNOPSIS

```
char *dm_bin_create_occur (variable, occurrence)
char  *variable;
int   occurrence;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine gets the specified occurrence from the variable. If the occurrence has not been allocated, this routine will allocate it. Note that ***occurrence*** must be less than or equal to the number of occurrences specified in the DBMS BINARY statement.

### RETURNS

0 if the variable is not found or the occurrence number is not valid
else a pointer to an occurrence in a binary variable

### VARIANTS

```
dm_bin_get_occur (variable, occurrence);
```

### RELATED FUNCTIONS

```
dbms BINARY variable[occ](length) [, variable [occ](length)...]]

dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

# dm_bin_delete_occur

## delete an occurrence in a binary variable

### SYNOPSIS

```
void dm_bin_delete_occur (variable, occurrence)
char  *variable;
int    occurrence;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY and the occurrence has been allocated, this routine frees the specified occurrence and sets the pointer to the occurrence to 0. If the occurrence has not been allocated, the routine does nothing.

### RETURNS

Nothing.

### RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

# dm_bin_get_dlength

## get the length of an occurrence in a binary variable

### SYNOPSIS

```
unsigned int dm_bin_get_dlength (variable, occurrence)
char  *variable;
int   occurrence;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY and the occurrence has been allocated, this routine returns the length of the contents in the specified occurrence.

### RETURNS

0 if variable or occurrence is not found,

else the length of the occurrence

### RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

# dm_bin_get_occur

## get the data in an occurrence of a binary variable

### SYNOPSIS

```
char *dm_bin_get_occur (variable, occurrence)
char  *variable;
int   occurrence;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY and the occurrence has been allocated, this routine gets the specified occurrence from the variable.

### RETURNS

0 if the variable or occurrence is not found
else a pointer to an occurrence in the variable

### VARIANTS

```
dm_bin_create_occur (variable, occurrence);
```

### RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

# dm_bin_length

## get the maximum length of an occurrence in a binary variable

### SYNOPSIS

```
unsigned int dm_bin_length (variable)
char  *variable;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine gets the maximum length of a single occurrence in the variable. To get the length of an occurrence's contents, use dm_bin_get_dlength.

### RETURNS

0 if the variable is not found
else the length of the variable

### RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

# dm_bin_max_occur

## get the maximum number of occurrences in a binary variable

### SYNOPSIS

```
int dm_bin_max_occur (variable)
char  *variable;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine gets the maximum number of occurrences in the variable.

### RETURNS

0 if variable is not found
else the number of occurrences in the variable.

### RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

# dm_bin_set_dlength

## set the length of an occurrence in a binary variable

### SYNOPSIS

```
void dm_bin_set_dlength (variable, occurrence, length)
char *variable;
int occurrence;
unsigned int length;
```

### DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine sets the maximum length of a single occurrence in the binary variable. ***length*** may be less than or greater than the variable's declared length.

### RETURNS

Nothing.

### RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
```

# dm_dbi_init

## initialize JAM for JAM/DB*i*

### SYNOPSIS

```
void dm_dbi_init ()
```

### DESCRIPTION

**JAM** must be initialized for use with **JAM**/DB*i*. This function tells **JAM** the class of error messages for **JAM**/DB*i* and how to handle the **JAM**/DB*i* JPL commands `dbms` and `sql`.

In the distributed source files `jmain.c` and `jxmain.c`, this function is called in the `initialize` routine. Developers modifying these source files or using a custom executive, may call this routine at another time. `dm_dbi_init` should be called before `sm_initcrt` to ensure that the message file is loaded properly.

### RETURNS

Nothing

# dm_dbms

## execute a DBMS command directly from C

### SYNOPSIS

```
int dm_dbms (arg)
char *arg;
```

### DESCRIPTION

Use this function to execute any DBMS command directly from C.

First ***arg*** is examined for the WITH ENGINE or WITH CONNECTION clause. If it is not used, dm_dbms assumes the default engine and connection. Next the colon preprocessor examines ***arg*** for colon variables. Finally, ***arg*** is passed to the appropriate routine for handing DBMS commands.

After executing the requested command, JAM/DB*i* updates all global status and error variables (@dm).

If the application has installed an entry function with DBMS ONENTRY, an exit function with DBMS ONEXIT, or an error handler with DBMS ONEXIT.the installed function will be called for commands executed through the function dm_dbms.

### RETURNS

0 is no error
else an error code from the default or installed error handler

### RELATED FUNCTIONS

dm_dbms_noexp (***arg***);

dm_sql (***arg***);

### EXAMPLE

```
int start_up ()
{
    int retcode;

    retcode = dm_dbms ("ONERROR CALL do_error");
    if (retcode)
    {
        sm_emsg ("Cannot install the application error handler.")
        return 0;
```

```
    }
    dm_dbms ("DECLARE c1 CONNECTION FOR USER :user PASSWORD :password");

    return 0;
}
```

# dm_dbms_noexp

## execute a DBMS command without colon preprocessing

### SYNOPSIS

```
int dm_dbms_noexp (arg)
char *arg;
```

### DESCRIPTION

This function is identical to `dm_dbms` except that colon preprocessing is NOT performed on ***arg***.

### RETURNS

0 is no error
else a return code from an installed or default error handler

### RELATED FUNCTIONS

```
dm_dbms (arg);

dm_expand (arg);

dm_sql (arg);

dm_sql_noexp (arg);
```

# dm_expand

## format a string for an engine

### SYNOPSIS

```
int dm_expand (engine, data, type, buf, buflen, edit)
char *arg;
char *data;
int type;
char *buf;
int *buflen;
char *edit;
```

### DESCRIPTION

Use this function to format a string for a particular engine and JAM type. The function copies the formatted string to a buffer provided by the program.

***engine*** is the name of an initialized engine. If this argument is null, JAM/DB*i* uses the default engine.

***data*** is the string to format. Use a JAM library functions such as `sm_getfield` to get the value of a field or LDB entry.

***type*** is one of the JAM types defined in `smedits.h`:

- `DT_CURRENCY`
- `DT_DATETIME`
- `DT_YESNO`
- `FT_CHAR`
- `FT_DOUBLE`
- `FT_INT`
- `FT_LONG`
- `FT_FLOAT`
- `FT_SHORT`

***buf*** is a buffer provided by the program. The program is responsible for allocating a buffer large enough for the formatted string. ***buflen*** points to the size of the buffer. Upon return from dm_expand, the value contained in the integer will be the length of the formatted text. The program can compare this value with the allocated length to ensure that truncation did not occur.

***edit*** is a date-time edit string describing ***data***. It is required when type is DT_DATETIME. Use sm_edit_ptr to get a format from a date-time field, or construct a format string using JAM 's date-time tokens. See sm_dtime for more information.

## RETURNS

0 is no error,
–1 if ***engine*** is invalid,
–2 if arguments are invalid (illegal **JAM** type, ***buflen*** <= 0, ***buf*** not allocated, or DT_DATETIME was used without a datetime edit)

–3 formatting routine failed

## RELATED FUNCTIONS

int dm_dbms_noexp (***arg***);

int dm_sql_noexp (***arg***);

## EXAMPLE

```
#include "smdefs.h"
#include "smedits.h"
#include "smerror.h"

#define   FLD_NOT_FOUND -1;
#define   MALLOC_ERROR  -2;
#define   EXPAND_ERROR  -3;
#define   NO_FORMAT     -4;


}int
formatter (src_name, dst_name, engine, jamtype)
char *src_name, *dst_name, *engine;
int jamtype;
{
   int dst_len, src_len, prec, ret;
   char *edit, *dst_buf, *src_buf;

            /* Get data. */
   /* Allocate a buffer based on the length of the source   */
   /* text and call getfield.                      */
   if ( (src_len = sm_n_dlength (src_name)) == -1)
      return FLD_NOT_FOUND;
```

```
    if ((src_buf=malloc(src_len + 1)) == 0)
        return MALLOC_ERROR;
    sm_n_getfield (src_buf, src_name);

    /* If no type was supplied, get it from the source field.*/
    if (jamtype == 0)
    {
        jamtype = sm_n_ftype(src_name, &prec);
    }

    /* If type is DT_DATETIME get format from field.   */
    if (jamtype == DT_DATETIME)
    {
        edit = sm_n_edit_ptr (src_name, UDATETIME);
        if (edit == 0)
        {
            edit = sm_n_edit_ptr (src_name, SDATETIME);
            if (edit == 0)
                return NO_FORMAT;
        }
        edit = edit +2;
    }

    /* Allocate a buffer based on the length of the
        desination field.*/
    if ( (dst_len = sm_n_length(dst_name)) == 0)
        return FLD_NOT_FOUND;
    if ((dst_buf=malloc(dst_len + 1)) == 0)
        return MALLOC_ERROR;

/* Call dm_expand.  */
    ret = dm_expand
          (engine, src_buf, jamtype, dst_buf, &dst_len, edit);
    if (ret == 0)
    {
        /* Write formatted text to destination field.      */
        sm_n_putfield (dst_name, dst_buf);
    }

        /* Free buffers.                          */
    free (src_buf);
    free (dst_buf);
```

```
/* If formatted string was too long for destination field,  */
/* ret will be greater than 0. If the format failed, it will */
/* be less than 0.                                           */
    return ret;
}
```

# dm_getdbitext

## get the text of the last executed dbms or sql command

### SYNOPSIS

```
char *dm_getdbitext
```

### DESCRIPTION

Use this function to get the full text of the last executed `dbms` or `sql` command. This includes all commands executed from JPL with `dbms` or `sql`, or executed from C with `dm_dbms`, `dm_dbms_noexp`, `dm_sql`, or `dm_sql_noexp`.

The text pointed to by the pointer returned by `dm_getdbitext` has a limited duration. If the application needs this information, it should call this function immediately after executing a JAM/DB*i* command. The program should process the returned string or copy it to a local variable before making additional function calls.

This is the same string that is passed to the first argument of an installed entry, error or exit handler, except that the error or exit handler is limited to 255 characters.

### RETURNS

A pointer to the last executed JAM/DB*i* command

### RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

`dbms ONEXIT [JPL` ***entrypoint*** `|` `CALL` ***function***`]`

### EXAMPLE

```
int
logfunc (stmt, engine, flag)
char *stmt;
char *engine;
int flag;
{
    FILE *fp;
    if (strlen(stmt) >= 255))
        stmt = dm_getdbitext();
    fp = fopen ("dbi.log", "a");
    fprintf (fp, "%s\n", stmt);
    fclose (fp);
    return 0;
}
```

# dm_init

## initialize JAM/DB*i* to access a specific database engine

### SYNOPSIS

```
int dm_init (engine, support_routine, options, arg)
char *engine;
int support_routine;
int options;
char *arg;
```

### DESCRIPTION

Before an application can access a database, JAM/DB*i* must perform an engine initialization. The initialization adds the engine name to the list of available engines, allocates a data structure for the engine, calls the engine's support routine to initialize the data structure, and sets case and error handling for the engine. Developers may use the `vendor_list` structure in `dbiinit.c` to initialize an engine at startup or else use `dm_init` to initialize an engine at a later point in the application.

The name for ***engine*** is chosen by the developer. If an application uses two or more engines, the application will use the mnemonic ***engine*** to indicate a particular DBMS. Most of the examples in the guide use a vendor name as the mnemonic, for example `sybase` or `oracle`, but any character string that is not a keyword is valid. Keywords are listed in Appendix A.. If ***engine*** is already installed, `dm_init` simply returns 0.

The name of ***support_routine*** is documented in the `dbiinit.c` file provided with the distribution. The file name is usually in the form `dm_`***vendor***`sup` where ***vendor*** is an abbreviated vendor name. Some examples are

- `dm_sybsup`
- `dm_orasup`
- `dm_intsup`

***options*** sets some defaults for the engine. It is composed of one or two flags: ***case*** and ***error***. They may be "or-ed."

The option ***case*** sets the case-handling feature of JAM/DB*i*. It determines how JAM/DB*i* uses case to map column names to JAM variables when executing a SELECT. The values are

| | |
|---|---|
| ■ `DM_DEFAULT_CASE` | Defaults to `DM_PRESERVE_CASE`. Another may be set by JYACC in the support routine. |

- `DM_PRESERVE_CASE` — Use case exactly as returned by the engine.
- `DM_FORCE_TO_UPPER_CASE` — Force all column names returned by an engine to upper case. Therefore, the application should use upper case names for JAM variables.
- `DM_FORCE_TO_LOWER_CASE` — Force all column names returned by an engine to lower case. Therefore, the application should use lower case names for JAM variables.

The option ***error*** sets the behavior of the default error handler. If none is given, the default is `DM_DEFAULT_DBI_MSG`. The values are

- DM_DEFAULT_DBI_MSG — Restrict the default error handler to using generic JAM/DB*i* messages for all error messages.
- DM_DEFAULT_ENG_MSG — Allow the default error handler to use engine-specific messages when an error occurs.

***arg*** is provided for future use. It should be set to 0.

Once the engine has been initialized, the application may declare a connection on it.

## RETURNS

0 if there is no error,
otherwise a return code from the support routine.

## RELATED FUNCTIONS

`dm_reset (`***name***`);`

## EXAMPLE

```
#include "dmerror.h"
#include "smusrdbi.h"

int retcode;
retcode = dm_init( "oracle",
                   dm_orasup,
                   DM_FORCE_TO_LOWER_CASE | DM_DEFAULT_DBI_MSG,
                   0);
```

# dm_reset

## disable support for a named engine

### SYNOPSIS

```
int dm_reset (name)
char *name;
```

### DESCRIPTION

An application may call this function to disable support for a named engine.

If the routine executes successfully, it performs the following steps:

1. Closes all active connections on the engine.
2. Calls the support routine to perform any engine-specific reset processing.
3. If ***name*** was the default engine, sets the default engine to 0.
4. Frees all data structures associated with the engine.

Once an engine has been reset, the application cannot connect to the engine unless it initializes the engine with `dm_init`.

### RETURNS

0 if the database engine was successfully disabled.

–1 if ***name*** was not a valid engine name.

### RELATED FUNCTIONS

```
dm_init (engine, support_routine, case, args);
```

### EXAMPLE

```
dm_reset ("oracle");
```

# dm_sql

## execute a SQL command directly from C

### SYNOPSIS

```
int dm_sql (arg)
char *arg;
```

### DESCRIPTION

Use this function to execute any SQL command directly from C.

First ***arg*** is examined for the WITH CONNECTION clause. If it is not used, `dm_sql` assumes the default connection. Next the colon preprocessor examines ***arg*** for colon variables. Finally, ***arg*** is passed to the appropriate routine for handing SQL commands.

After executing the requested command, JAM/DB*i* updates all global status and error variables (`@dm`).

If the application has installed an entry function with DBMS ONENTRY, an exit function with DBMS ONEXIT, or an error handler with DBMS ONEXIT.the installed function will be called for commands executed through the function `dm_sql`.

### RETURNS

0 is no error,
else the return code from the default or an installed error handler

### RELATED FUNCTIONS

```
int dm_dbms (arg);
```

### EXAMPLE

```
int select_ssn ()
{
    int retcode;
    retcode = dm_sql ("SELECT * FROM emp WHERE ssn LIKE :+ssn");
    return retcode;
}
```

# dm_sql_noexp

## execute a SQL command without colon preprocessing

### SYNOPSIS

```
int dm_sql_noexp (arg)
char *arg;
```

### DESCRIPTION

This function is identical to `dm_sql` except that colon preprocessing is NOT performed on ***arg***.

### RETURNS

0 is no error,
else an code from the default or an installed error handler

### RELATED FUNCTIONS

```
int dm_dbms (arg);

int dm_dbms_noexp (arg);

int dm_expand (arg);

int dm_sql (arg);
```

# Chapter 16. JAM/DB*i* Utility Reference

Unlike the JAM utilities, the JAM/DB*i* utilities `f2tbl` and `tbl2f` are not distributed as executables. Libraries, object code, and a makefile for `f2tbl` and `tbl2f` are included with the JAM/DB*i* distribution. Developers must edit the makefile to describe the environment and to supply the paths to the JAM, JAM/DB*i*, and database installations.

The rest of this chapter contains reference pages for the JAM/DB*i* utilities:

- `f2tbl`: create a database table from a JAM form
- `tbl2f`: create a JAM form from a database table

Each reference page has the following sections:

- A synopsis of the utility, including a listing of options and arguments.
- A description of the utility.
- Examples.

# f2tbl

## create a database table from a JAM form

### SYNOPSIS

```
f2tbl [-i] \
   [-u user [-p password]] [-s server] [-d database] [-y dictionary] \
   [-t tablename] [-l{l|u}] [-c outfile] [-f] screen ...
```

### OPTIONS

- `-i` Run utility in interactive mode. This opens windows where you may enter any information not supplied on the command line.
- `-u` Connect with the given user name.
- `-p` Connect with the given password.
- `-s` Connect to the named server.
- `-d` Connect to the named database.
- `-y` Connect using the named dictionary.
- `-t` Use ***tablename*** as the name of the database table. By default, the table name is the screen name minus `SM_FEXTENSION`.
- `-l` Convert all field names to lower or upper case column names in the CREATE statement. For case, use `-ll` for lower or `-lu` for upper. The default is to use the case of the field names.
- `-c` Write the SQL CREATE statement(s) to the named ASCII file. Do not create the table on the database.
- `-f` Overwrite an existing database table or script file. To overwrite an existing table, `f2tbl` executes a SQL statement to drop the existing table before creating the new one. **All rows in the old table will be lost when the table is dropped.**

If no options or invalid options are given, the utility displays a usage message and a list of the valid options.

### DESCRIPTION

Use this utility to create a database table or a script file for one or more JAM screens. If you are converting many screens, interactive mode is recommended.

For each screen, the utility defines a table with a column for each named field on the screen. The column's datatype is engine-specific and is based on the field's **JAM** type. If a field has a character **JAM** type, the utility calculates the column length by examining the field's edits. Based on the field's null field edit, the utility declares whether or not the column accepts nulls.

The `-c` flag is recommended, particularly for new users. With this flag, `f2tbl` writes the CREATE statement to an ASCII file where it may be examined and edited before it is executed.

Some of the logon flags are not supported on some engines. If you use an unsupported logon flag, the utility ignores it and the argument. See the engine-specific *Notes* for a list of the supported logon options.

If `f2tbl` cannot create the table, it displays either a **JAM**/DB*i* or engine error message.

## Converting Fields to Column Definitions

### COLUMN NAME

`f2tbl` uses the field name as the column name. If a field is unnamed, `f2tbl` ignores it. Please note that some valid **JAM** field names may be not be valid column names. For example, **JAM** allows the characters `$` and `.` in **JAM** field names but many engines do not permit these characters in column names. If a name is illegal, `f2tbl` will display the engine's error message when it attempts to create the table.

### MATCHING A JAM TYPE TO AN ENGINE DATATYPE

A field has exactly one **JAM** type. Since a field may have more than one of the qualifying PF4 characteristics, **JAM** uses precedence rules when determining the **JAM** type. You may determine a field's **JAM** type by looking at its summary screen while inside the Screen Editor.

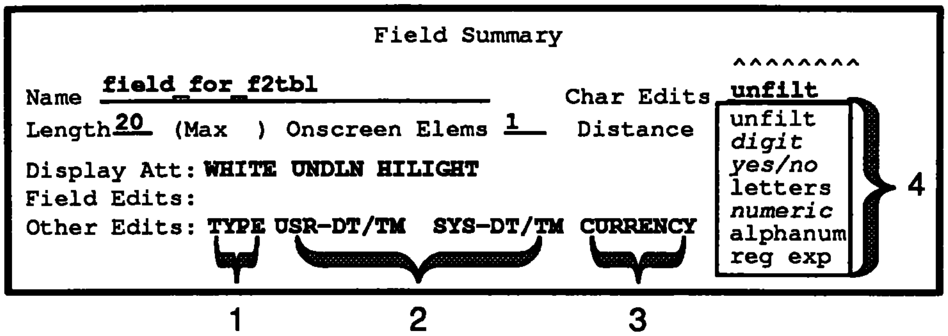



Figure 32: Field Summary Window (PF5 in draw mode). Use the summary screen to determine a field's JAM type. A TYPE edit has the highest priority, then a date time edit, then a currency edit, and finally a character edit.

| Summary Keyword | Setting of Field Characteristic (PF4 menu in draw mode) | Submenu Option | JAM Type |
|---|---|---|---|
| TYPE | type<br>(C types for structures) | char string<br>int<br>unsigned int<br>short int<br>long int<br>float<br>double<br>zoned dec.<br>packed dec. | FT_CHAR<br>FT_INT<br>FT_UNSIGNED<br>FT_SHORT<br>FT_LONG<br>FT_FLOAT<br>FT_DOUBLE<br>FT_ZONED<br>FT_PACKED |
| USR-DT/TM<br>SYS-DT/TM | misc. edits | date or time | DT_DATETIME |
| CURRENCY | misc. edits | currency | DT_CURRENCY |
| Char Edits | char edits | digits only<br>yes/no field<br>numeric | FT_UNSIGNED<br>DT_YESNO<br>FT_DOUBLE |

Figure 33: The keywords on the summary window indicate which of the field characteristics has set the field's JAM type.

If the word `TYPE` appears on the field summary window, you must press the `PF4` key and choose `type` to open the C type submenu. The setting on the submenu indicates the JAM type. For example, if `double` is chosen on the submenu, the JAM type is `FT_DOUBLE`. Figure 33 shows the C type names and the corresponding JAM types.

If the keyword TYPE is not on the summary window, the **JAM** type is immediately determinable. With the keyword USR-DT/TM or SYS-DT/TM, the **JAM** type is DT_DATETIME. Otherwise, with the keyword CURRENCY, the **JAM** type is DT_CURRENCY. If none of those keywords appear, the character edit may apply: with digits only the **JAM** type is FT_UNSIGNED, with yes/no field the type is DT_YESNO, or with numeric the type is FT_DOUBLE.

If none of the above edits are set, but the field has a word-wrapped edit, the **JAM** type is FT_VARCHAR. For all other fields, the **JAM** type is FT_CHAR.

Since engines uses different names for datatypes, the mapping of **JAM** types to engine datatypes is listed in the engine-specific *Notes*.

## CALCULATING THE COLUMN LENGTH

If the field has the **JAM** type FT_CHAR, FT_VARCHAR, or DT_YESNO, f2tbl attempts to use the field's length as the column length. For all other **JAM** types, a length is not calculated because the **JAM** type is mapped to an engine datatype with a default length.

For FT_VARCHAR fields (word-wrapped), the calculated length is the maximum shifting length times the total number of occurrences in the array. For FT_CHAR and DT_YESNO fields, the calculated length is the maximum (shifting) length of the field.

If the calculated length in either case is greater than the length permitted by the engine, f2tbl will use the maximum permitted length.

## DEFINING A COLUMN AS NULL OR NOT NULL

If the field has a null field edit, the column is defined as permitting nulls. On some engines, this is the default. Others may explicitly use the keyword NULL.

If the field does not have a null field edit, the column is defined as NOT NULL.

## OUTPUT

f2tbl builds a SQL CREATE statement in a form similar to the following:

```
CREATE TABLE tablename (
    column_1 datatype [(length)] [NOT] [NULL] ,
    column_2 datatype [(length)] [NOT] [NULL] ,
    ...
    column_n datatype [(length)] [NOT] NULL]
)
```

## Example

Assume the screen named `inventory` has the following named fields:

- `id_no`
- `product_name`
- `price`
- `description`

The figures below show the field summary window for each field. A sample column declaration is also shown for each field. Since column datatypes are engine-specific, the names used here are solely for illustration.

```
                         Field Summary
                                                       ^^^^^^^^
Name id_no________________________          Char Edits digit___
Length11_ (Max   ) Onscreen Elems 3_  Distance  (Max Occurs 15 )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: TYPE
```

Figure 34: Field `id_no`. The type edit is set to `char string` to override the digits only character edit. Therefore, its JAM type is `FT_CHAR`.

The column definition would appear like the following

```
id_no char (11)    NOT NULL
```

Since the field does not have a word-wrapped edit, the number of occurrences is ignored. In addition, since the field does not have a null field edit, the column is defined as not allowing null values.

```
                         Field Summary
                                                  ^^^^^^^^
Name product_name_______________      Char Edits unfilt__
Length15_ (Max25) Onscreen Elems 1_   Distance  (Max Occurs 15 )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits:
```

Figure 35: Field `product_name`. Its **JAM** type is `FT_CHAR`.

The column definition would appear like the following

```
product_name    char (25) NOT NULL
```

Note that the column length is 25 which is the maximum shifting length, rather than 15 which is the onscreen length. Since the field does not have a word-wrapped edit, the number of occurrences is ignored. In addition, since the field does not have a null field edit, the column is defined as not allowing null values.

```
                         Field Summary
                                                  ^^^^^^^^
Name price______________________      Char Edits numeric_
Length11_ (Max   ) Onscreen Elems 1_  Distance  (Max Occurs15 )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: CURRENCY
```

Figure 36: Field `price`. Its **JAM** type is `DT_CURRENCY`.

If the engine had a datatype called `money`, the column definition would appear like the following

```
price money     NOT NULL
```

On most engines, a currency datatype has a predefined length. In this case, `f2tbl` ignores the field's length. If the engine does not have a currency type, `f2tbl` may use a type such as `NUMERIC` or `FLOAT` and it may calculate a length or precision.

Since the field does not have a null field edit, the column is defined as not allowing null values.

```
                         Field Summary
                                                        ^^^^^^^^
Name description                          Char Edits unfilt
Length 50  (Max  ) Onscreen Elems 5    Distance  (Max Occurs   )
Display Att: WHITE UNDLN HILIGHT
Field Edits: WDWRP
Other Edits: NULL
```

Figure 37: Field `description`. Its **JAM** type is `FT_VARCHAR`.

The column definition would appear like the following

```
description char (250)
```

Note that the column's length is the field's length 50 multiplied by the number of elements 5, and therefore 250. In this case, the field's number of occurrences affected the column length because the word-wrap edit was set. Since the field also has a null field edit, the column is defined as permitting null values. Some engines may also use the keyword `NULL` at the end of the definition.

The resulting CREATE statement would appear similar to the following:

```
CREATE TABLE inventory (
   id_no ( 11 ) NOT NULL,
   product_name char ( 20 ) NOT NULL,
   price money NOT NULL,
   description char ( 250 )
)
```

## SEE ALSO

Engine-specific *Notes*

# tbl2f

## create a JAM screen from a database table

### SYNOPSIS

```
tbl2f [-i] \
  [-u user [-p password]] [-s server] [-d database] [-y dictionary] \
  [-j jpl_template] [-t screen_template] \
  [-k index_key] [-l {u|l}] [-e ext] [-f] table [table...]
```

### OPTIONS

| | |
|---|---|
| -i | Run utility in interactive mode. This opens windows where you may enter any information not supplied on the command line. |
| -u | Connect with the given user name. |
| -p | Connect with the given password. |
| -s | Connect to the named server. |
| -d | Connect to the named database. |
| -y | Connect using the named dictionary. |
| -j | Use the named file as a template for creating the JPL screen module and assigning control strings. The utility looks in the current directory and in the `SMPATH` directories for the named file. The default template is ***db***`exm.jpl`. |
| -t | Use the named file as a template for creating the **JAM** screen. |
| -k | Use the named column as the index key in the JPL procedures. If this flag is not, `tbl2f` chooses an index by querying the engine's system tables. If it cannot find one for the table, it defaults to the first column in the table. |
| -l | Force the case of column names in the JPL procedures and the field names on the screen to upper (`-lu`) or lower (`-ll`) case. The default is to preserve case. |
| -e | Append ***ext*** as the extension to the screen files. The default is `SMFEXTENSION`, typically `JAM`. |
| -f | Overwrite an existing screen file. |

If no options or invalid options are given, the utility displays a usage message and a list of the valid options.

### DESCRIPTION

Use this utility to create a **JAM** screen for each named database table. If you are converting many tables, interactive mode is recommended.

In each screen, `tbl2f` will create the following

- A field for each column in the table, with up to 250 fields created in total.
- Display text on the screen identifying the name of the screen and the name of each field.
- Control strings to call the JPL procedures.
- JPL procedures to query and update the table.

The following topics are covered in the remaining sections:

- Controlling the case of field names and predicting field characteristics on the created screen (page 214).
- Using a JPL template file to change and add procedures in the JPL screen module (page 216).
- Using a JPL template file to put control strings on the created screen (page 223).
- Using a screen template to change the default screen characteristics (page 225).

## Fields

The utility creates a field for each column in the table, with up to 250 fields created in total. Field characteristics are assigned according to the column's data type. A field is named for its column in the table. The field's length is taken from the column definition.

### FIELD NAMES

When `tbl2f` creates a field, it names the field for a database column. By default, the utility uses case exactly as returned by the database. On engines where column names are always upper case, for example ORACLE, the utility will create upper case field names by default. On engines where columns names may be in either or mixed case, the utility will create field names using the exact case of the column name.

The utility provides the option of forcing case to upper or lower. This is done with the `-l` flag on the command line or with the `Options` menu in interactive mode. Please note that this option forces the case of both onscreen field names and the column names used in the SQL statements in the JPL procedures.

To the left of each field, the utility displays the field name. Note that if the field name contains any draw field symbols, such as the underscore, those characters will be converted to fields when the screen is edited.

While almost all column names are valid JAM identifiers, `tbl2f` does not verify if a column name is a valid JAM field name and thus does not report an error for bad field names.

You may easily verify the validity of field names by using the **JAM** utility `f2asc` to create an ASCII version of the screen file and then run `f2asc` to convert it back to binary. Since `f2asc` validates field names before re-creating the binary file, it will report any invalid field names. If it does, you may use a text editor to quickly edit the `f2asc` ASCII file and then convert the file to a binary screen file. If the screen has JPL procedures referencing the fields, you should change only the references to the invalid field name and not change the references to the column name. For example, if the table `inventory` contained three columns `id#`, `product`, and `description`, the field names `product` and `description` are valid, but the field name `id#` is not. If the field were renamed `id_no`, a JPL statement like the following

```
sql SELECT id#, product, description FROM inventory \
   WHERE id# = :+id#
```

should be edited to

```
dbms ALIAS id# id_no
sql SELECT id#, product, description FROM inventory \
   WHERE id# = :+id_no
```

## FIELD CHARACTERISTICS

When `tbl2f` creates a field, it assigns field characteristics based on the column's datatype and characteristics. The distributed JPL file ***db***`t2f.jpl`, where ***db*** is an abbreviated vendor name, equates engine datatypes with **JAM** types. For example, an engine datatype such as `money` is typically treated as the **JAM** type `DT_CURRENCY`. An engine datatype `char` is usually treated as the **JAM** type `FT_CHAR`. See the engine-specific *Notes* for a listing.

Based on the **JAM** type, the field is assigned the following edits:

| Column Type Equivalent to: | Assigned Field Characteristics: | | |
|---|---|---|---|
| JAM Type | C type (non-default) | misc. edits | char edits |
| FT_SHORT | short int | | digits only |
| FT_INT | int | | digits only |
| FT_UNSIGNED | unsigned int | | digits only |
| FT_LONG | long int | | digits only |
| FT_FLOAT | float | | numeric |
| FT_DOUBLE | double | | numeric |
| DT_DATETIME | | date time | unfiltered |
| DT_CURRENCY | | currency | unfiltered |
| FT_CHAR | | | unfiltered |
| FT_VARCHAR | | | unfiltered |

Since engines uses different names for datatypes, the mapping of datatypes to JAM types is listed in the engine-specific *Notes*.

The length of the field depends on the field's JAM type.

- An `FT_CHAR` or `FT_VARCHAR` field is assigned the length of the column, up to the maximum length of 255.
- A `DT_DATETIME` column is assigned a default length of 20.
- A numeric type column is assigned an engine-specific length and precision defined in ***db***`t2f.jpl`.

`tbl2f` supports the engine's standard datatypes. Some engines permit developers to define their own datatypes. To change the JAM type of a standard datatype or to supply one for a user datatype, you must modify ***db***`t2f.jpl`. After editing the file, you must recompile the `tbl2f` executable so that the new assignments are used.

## JPL Procedures

As a part of the distribution, JAM/DB*i* supplies a template of JPL procedures. It uses this template to create a JPL screen module. The default template ***db***`exm.jpl` builds procedures to fetch, update, insert, and delete rows in the table.

These table-specific procedures are created with the use of special `tbl2f` variables which begin with a double colon (`::`). The `tbl2f` variables provide strings or statements to help perform some commonly useful tasks.

There are 18 `tbl2f` variables. The variable names are composed of a root and a suffix. The 6-character root describes an action such as `::CLR_` for clear or `::QBEX` for query by expression. The 3-character suffix describes which columns the action will involve. The roots and suffixes are described in the tables below.

| *Root* | *Description* |
|---|---|
| `::CLR_` | for clearing the onscreen value of one or more columns in the form<br>`cat column` |
| `::COND` | for a list of conditions in the form<br>`column = :+column [:CONAND column = :+column ...]` |
| `::LIST` | for a list of one or more column names in the form<br>`column [:LISTAND column ...]` |
| `::SET_` | for a list of one or more onscreen column values in the form<br>`:+column [:SET_AND :+column ...]` |
| `::VAL_` | for a list of one or more onscreen column values in the form<br>`:+column [:VAL_AND :+column ...]`<br>on some engines this is equivalent to `SET_` |
| `::QBEX` | for `if` block(s) that build a query-by-expression clause in the form<br>`if (column != "")`<br>`{`<br>`CAT QBYEXAM QBEXAM VAND "column {:LIKEWORD\|=} :+column "`<br>`CAT VAND LIKEAND`<br>`}` |

| *Suffix* | *Description* |
|---|---|
| `ALL` | use all columns |
| `EIN` | use all columns except the index column |
| `IND` | use only the index column |

Every combination of ***rootsuffix*** is a legal `tbl2f` variable.

If there any other double colon variables in the template, `tbl2f` simply strips off the first column. The utility will attempt to expand standard colon variables. If

```
:tabname
```

is used in the template, the utility replaces it with the name of the table that it is processing. If it cannot expand a colon variable it ignores it. For best results, use the backslash to preserve all variables that should be expanded by the application rather than the utility. For example,

```
# tbl2f will replace :tabname with the table name
sql SELECT * FROM :tabname

# JPL will replace :uid when the application is run
dbms DECLARE conn1 CONNECTION FOR USER \:uid
```

The sections below give an example for each root showing a suggested use in a template and its output. The output is shown in two forms, one generic and the other based on a sample table called `acc`. The table `acc` contains three columns:

- `ssn` a character column of length 11
- `salary` a money column
- `exmp` an integer column

The index column for `acc` is `ssn`.

## ::CLR_ VARIABLES

Use the `::CLR_` variables to create `cat` statements to clear one, some, or all the onscreen column values.

**Syntax in a JPL Template**

```
proc clear
::CLR_ALL
return
```

**Output Syntax in a JPL Screen Module**

```
proc clear
cat index_field
cat field1
cat field2
...
return
```

**Output for Sample Table** `acc`

```
proc clear
cat ssn
cat salary
cat exmp
return
```

## ::COND VARIABLES

Use a ::COND variable to get a string suitable for a WHERE clause. While all ::COND variables are legal, the condition ::CONDALL or ::CONDIND is more useful than ::CONDEIN when performing a SELECT, UPDATE, or DELETE.

If ::CONDALL is used, the conditions are separated with the JPL variable :CONDAND. In the distributed templates, CONDAND is usually the keyword AND.

**Syntax in a JPL Template**

```
sql SELECT * FROM :tabname WHERE ::CONDIND
```

**Output Syntax in JPL Screen Module**

sql SELECT * FROM ***table*** WHERE ***index_column*** = :+***index_field***

**Output for Sample Table** acc

```
sql SELECT * FROM acc WHERE ssn = :+ssn
```

## ::LIST VARIABLES

Use a ::LIST variable to get a string of one, some, or all column names separated by the value of the JPL variable LISTAND. In the distributed template, LISTAND is usually a comma.

**Syntax in a JPL Template**

```
vars LISTAND(10)
cat LISTAND ", "

sql SELECT ::LISTALL FROM :tabname
```

**Output Syntax in a JPL Screen Module**

vars LISTAND(10)<br>
cat LISTAND ", "

sql SELECT ***column1*** :LISTAND ***column2*** ... FROM ***table***

**Output for Sample Table** acc

```
vars LISTAND(10)
cat LISTAND ", "

sql SELECT ssn :LISTAND salary :LISTAND exmp FROM acc
```

## ::QBEX VARIABLES

Use a ::QBEX variable to create JPL statements which at runtime generate a string expression suitable for the WHERE clause of a query-by-expression procedure. For each column requested by the suffix, it creates a block of statements which test if the onscreen field is empty and concatenate a JPL variable called QBYEXAM with the name of the column and its onscreen value. Other procedures may use the value of QBYEXAM as the search criteria for a SELECT or an UPDATE.

**Syntax in a JPL Template**

```
vars QYBEXAM LIKEWORD(10) LIKEAND(10)
cat LIKEWORD "LIKE"
cat LIKEAND "AND"

proc sellike
# Call procedure "query" to build the QBE expression
# QBYEXAM is replaced when the application is executed.
jpl query
sql SELECT * FROM :tabname \:QYBEXAM
return

proc query
# Assign a value to the JPL variable "QBYEXAM"
vars VAND(10)
cat QYBEXAM
cat VAND
# ::QBEXALL puts an "if" block for each column here:
###########################################################
::QBEXALL
###########################################################
if (QBYEXAM != "")
{
   cat QBEXAM " WHERE " QYEXAM
}
return 0
```

**Output Syntax in a JPL Screen Module**

For each FT_CHAR column, ::QBYEXAM produces the following statements:

```
if (field != "")
{
   cat QBYEXAM QBEXAM VAND "column :LIKEWORD :+field"
   cat VAND LIKEAND
}
```

For each non-FT_CHAR column (e.g. numeric and date columns), QBYEXAM produces the following statements:

```
if (field != "")
{
   cat QBYEXAM QBEXAM VAND "column = :+field"
   cat VAND LIKEAND
}
```

**Output for Sample Table** acc

```
vars QYBEXAM LIKEWORD(10) LIKEAND(10)
cat LIKEWORD "LIKE"
cat LIKEAND "AND"

proc sellike
# Call procedure "query" to build the QBE expression
jpl query
sql SELECT * FROM acc :QBYEXAM
return

proc query
# Assign a value to the JPL variable "QBYEXAM"
cat QYBEXAM
cat VAND
# ::QBYEXAM puts an "if" block for each column here:
##########################################################
if (ssn != "")
{
   cat QBYEXAM QYEXAM VAND " ssn :LIKEWORD :+ssn"
   cat VAND LIKEAND
}
if (salary != "")
{
   cat QBYEXAM QYEXAM VAND " salary = :+salary"
   cat VAND LIKEAND
}
if (exmp != "")
{
   cat QBYEXAM QYEXAM VAND " exmp = :+exmp"
   cat VAND LIKEAND
}
##########################################################
```

```
if (QBYEXAM != "")
{
   cat QBEXAM " WHERE " QYEXAM
}
return 0
```

## ::SET_ VARIABLES

Use a ::SET_ variable to get a string of the name and onscreen value of one or more columns. The pairs of column name and column value are separated by the value of the variable SET_AND. In the distributed template SET_AND is a usually comma. These variables are useful for the SET clause of an UPDATE statement.

**Syntax In a JPL Template**

```
vars SET_AND
cat SET_AND ","

sql UPDATE :tabname SET ::SET_EIN WHERE ...
```

**Output Syntax In a JPL Screen Module**

```
vars SET_AND
cat SET_AND ","

sql UPDATE table SET column1 = :+column :SET_AND \
   column2 = :+column2 ... WHERE ...
```

**Output for Sample Table** acc

```
vars SET_AND
cat SET_AND ","

sql UPDATE acc SET salary = :+salary :SET_AND \
   exmp = :+exmp WHERE ...
```

## ::VAL_ VARIABLES

Use a ::VAL_ variable to create a string of the name and onscreen value of one or more columns. The pairs of column name and column value are separated by the value of the variable VAL_AND. In the distributed template VAL_AND is a usually comma. These variables are useful for the VALUES clause of an ONSERT statement. In the distributed template, VAL_AND is a comma.

**Syntax in a JPL Template**

```
vars LISTAND(10) VAL_AND(10)
cat LISTAND ", "
cat VAL_AND ", "

sql INSERT INTO :tabname (::LISTALL) \
   VALUES (::VAL_ALL)
```

**Output Syntax in a JPL Screen Module**

```
vars LISTAND(10) VAL_AND(10)
cat LISTAND ", "
cat VAL_AND ", "

sql INSERT INTO table (column1 :LISTAND column2 ...) \
   VALUES (:+column1 :VAL_AND :+column2 ...)
```

**Output for Sample Table** acc

```
vars LISTAND(10) VAL_AND(10)
cat LISTAND ", "
cat VAL_AND ", "

sql INSERT INTO acc (ssn :LISTAND salary :LISTAND exmp) \
   VALUES (:+ssn :VAL_AND :+salary :VAL_AND :+exmp)
```

## Control Strings

If a screen template is not used, control strings may be assigned to logical keys PF1–PF10, and SPF1–SPF10 using the JPL template. The syntax is

```
#jctl 1     control string for PF1
#jctl 2     control string for PF2
#jctl 3     control string for PF3
#jctl 4     control string for PF4
#jctl 5     control string for PF5
#jctl 6     control string for PF6
#jctl 7     control string for PF7
#jctl 8     control string for PF8
#jctl 9     control string for PF9
#jctl 10    control string for PF10
#jctl 11    control string for SPF1
#jctl 12    control string for SPF2
#jctl 13    control string for PF13
#jctl 14    control string for PF14
#jctl 15    control string for PF15
```

```
#jctl 16    control string for PF16
#jctl 17    control string for PF17
#jctl 18    control string for PF18
#jctl 19    control string for PF19
#jctl 20    control string for SPF10
```

Note that the pound sign must be in the first column of the line and the word `jtcl` must follow it immediately. Any lines that do not begin this way are assumed to be JPL comments and they are simply copied to the JPL screen module. ***controls string*** may be any valid JAM control string. Control strings are documented in the *Author's Guide* of the JAM manual.

You may assign none, some, or all these control strings. No assignments are made for numbers outside the range of 1 to 20. The assignments may be in any order and place in the template but we recommend that you put them in a block at the beginning of the template. If the template assigns a control string more than once, the last assignment takes precedence.

In the JPL template you may wish to include a procedure that displays a status line message describing the key assignments. Remember that `%K` may be used in messages to display keytop labels. See the *JPL Guide* for more information on message display.

If a screen template is used (`-t` option), `tbl2f` ignores any control string assignments in the JPL template.

**Example Template**

```
#jctl 1     ^jpl select_all
#jctl 2     ^jpl select_by_index
#jctl 10    main_menu

proc message_line
msg setbkstat \
   "%KPF1: Select_All  "\
   "%KPF2: Select_by_Index  "\
   "%KPF10: Main Menu"
return

proc select_all
vars LISTAND(10)
cat LISTAND ","
sql SELECT ::LISTALL FROM :tabname
return

proc select_by_index
sql SELECT ::LISTALL FROM :tabname WHERE ::CONDIND
return
```

## Screen Characteristics

An existing **JAM** screen may be used as a template for new screens created with `tbl2f`. A screen template is supplied with the `-t` flag or in interactive mode. If you are using a local engine on a PC, you may not have enought memory to use a screen template.

The following screen characteristics are supported by the template:

1. *Minimum number of lines and columns.* `tbl2f` will not create a screen smaller than these dimensions. If necessary, it may create a larger screen. The maximum width is the default number of columns defined in the video file. If a field is longer than the onscreen width, `f2tbl` creates a shifting field. If there are not enough onscreen lines for the fields, `tbl2f` creates a virtual screen with up to the maximum 254 lines.
2. *Border style and attribute.* `tbl2f` uses the template's border style and attribute for the new screen.
3. *Background color.* `tbl2f` uses the template's background color for the new screen.
4. *Start as menu setting.* If the template screen has menu fields, set the starting mode for the new screen.
5. *Screen-level help.* Assign a screen-level help window for the new screen.
6. *Screen entry/exit functions.* Assign screen entry and exit hook functions for the new screen.
7. *Screen-level keyset.* Assign a keyset for the screen.
8. *Display text attribute.* Use the PF4 key in draw mode to set the attributes for `pen` on the template screen. `tbl2f` will use this pen when writing labels on the new screen.

Please note that any JPL in the screen JPL module of the template is not copied to the new screen. Use the JPL template option to supply JPL procedures for the screen.

`tbl2f` has its own default attributes for the fields it creates. Any draw field symbols on the template screen are copied to the new screen, but they are not used by the utility.

All control strings on the template screen are copied to the new screen. Any control string assignments in the JPL template are ignored.

All fields and display text on the template are written to the new screen. `tbl2f` begins writing the database fields at the first empty line below the template's display text and/or fields. The current release does not copy groups from the template.

### SEE ALSO

Engine-specific *Notes*

# Appendixes

# Appendix A.
# Keywords

Below is a list of all the keywords for JAM/DB*i* commands. We strongly encourage developers to avoid using these keywords as identifiers, particularly for cursors, connections, engines, and transactions. We also recommend that developers avoid using these keywords when naming JAM variables which will be used in a `dbms` or `sql` statement. The list includes keywords supported by Release 4.8 and Release 5.

```
alias
autocommit

begin
binary
browse

call
cancel
catquery
checkpt_interval
close
close_all_connections
commit
connect
connected
connection
continue
continue_bottom
continue_down
continue_top
continue_up
create_proc
create_trigger
count
current

cursor
cursors

database
db
dbms
declare
disconnect
drop_proc
drop_trigger

end
engine
error
error_continue
exec
execute
execute_all

flush
file
for
format

heading

interfaces

jpl

locklevel
locktimeout
logon
logoff

max

next
null

occur
off
on
onentry
onerror
onexit
options
out
output

password
prepare_commit
print
proc
proc_control
```

redirect
return
retvar
rfjournal
rollback
rpc

save
schema
select
select_aliases
separator
serial

server
set
set_buffer
single_step
sql
start
stop
stop_at_fetch
store
supreps

tee

timeout
to
transaction
type

unique
update
use
user
using

warn
with

# Appendix B.
# Error and Status Codes

Like JAM, JAM/DB*i* uses symbolic constants to define its error codes. Any error handling functions written in C may simply include the header file `dmerror.h` to use these constants. JPL, on the other hand, is an interpreted language and it has no access to these constants when performing variable substitution. JPL does have access, however, to constants in the local data block (LDB). Therefore, we recommend that developers using JPL for error handling also use the data dictionary and an initialization file to define all the constants that the procedures will need. A sample data dictionary and initialization file are provided with the JAM/DB*i* distribution. Please see the README for directions on using these samples.

For example, if a JPL procedure must test for the no more rows signal, add the entry DM_NO_MORE_ROWS to the data dictionary, with length 5 and scope 1. Use an initialization file such as `const.ini` to assign its value,

```
"DM_NO_MORE_ROWS" "53256"
```

The developer may then use the name of the LDB constant in JPL procedures rather than hard-coding the decimal value in the procedure. For example, it may execute the following

```
proc select_all
sql SELECT * FROM emp
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg "All rows returned."
```

The table lists the constant's name, its decimal value, and its default error message.

| *Constant* | *Value* | *Message* |
|---|---|---|
| DM_NODATABASE | 53249 | No database selected. |
| | | |
| DM_NOTLOGGEDON | 53250 | Not logged in. |
| DM_ALREADY_ON | 53251 | Already logged on. |
| DM_ARGS_NEEDED | 53252 | Arguments required. |
| DM_LOGON_DENIED | 53253 | Logon denied. |
| DM_BAD_ARGS | 53254 | Bad arguments. |
| DM_BAD_CMD | 53255 | Bad command. |
| DM_NO_MORE_ROWS | 53256 | No more rows indicator. |
| DM_ABORTED | 53257 | Processing aborted due to DB error. |
| DM_NO_CURSOR | 53258 | Cursor does not exist. |
| DM_MANY_CURSORS | 53259 | Too many cursors. |
| | | |
| DM_KEYWORD | 53260 | Bad or missing keyword. |
| DM_INVALID_DATE | 53261 | Invalid date. |
| DM_COMMIT | 53262 | Commit failed. |
| DM_ROLLBACK | 53263 | Rollback failed. |
| DM_PARSE_ERROR | 53264 | SQL parse error. |
| DM_BIND_COUNT | 53265 | Incorrect number of bind vars. |
| DM_BIND_VAR | 53266 | Bad or missing bind variable. |
| DM_DESC_COL | 53267 | Describe select column error. |
| DM_FETCH | 53268 | Error during fetch. |
| DM_NO_NAME | 53269 | No name specified. |

| *Constant* | *Value* | *Message* |
|---|---|---|
| DM_END_OF_PROC | 53270 | End of procedure. |
| DM_NOCONNECTION | 53271 | No connection active. |
| DM_NOTSUPPORTED | 53272 | Command not supported for the specified engine. |
| DM_TRAN_PEND | 53273 | Transaction pending. |
| DM_NO_TRANSACTION | 53274 | Transaction does not exist. |
| DM_ALREADY_INIT | 53275 | Engine already installed. |

# Appendix C. Fields in a JAM/DB*i* Application

JAM/DB*i* applications primarily use fields to move data between the end user and a database. Developers create a named JAM field for each database column that the end user will view or update.

In this chapter, we give some suggestions on creating fields for a JAM/DB*i* application. We briefly discuss how you may use the various field settings of JAM's PF4 key when creating JAM/DB*i* fields, and how these settings may affect an application. In particular, we discuss how these settings affect

- the end user's interface
- data formatting between JAM and a database

The physical flow of data between JAM and a database is discussed in detail in Chapters 8. and 9..

## 22.1. JAM FIELD CHARACTERISTICS (PF4)

JAM's field characteristics provide developers with many tools for creating attractive and successful interfaces. Very briefly, we highlight here those features that are likely to be useful to JAM/DB*i* developers.

Furthermore, we discuss how the features affect data formatting between JAM and an engine.

22.1.1.
## Field Display Attributes

The use of display attributes like color or highlight have no effect on the data.

22.1.2.
## Character Edits

A character edit provides one way of helping end users enter data quickly and correctly, since it verifies each character as it is entered.

Developers may use character edits to enforce rules or checks at the application frontend. Although rules and data integrity should still be enforced by the database, effective use of character edits should reduce the number of unnecessary trips to the server, thus improving the application's efficiency.

Embedded punctuation is a useful feature with certain character edits. When a field has the character edit digits-only, letters-only, or alphanumeric the developer may save punctuation characters in the field which the user cannot type over or delete. For example, a field that accepts a U.S. telephone number would have a digits-only character edit and parentheses and a dash as embedded punctuation.

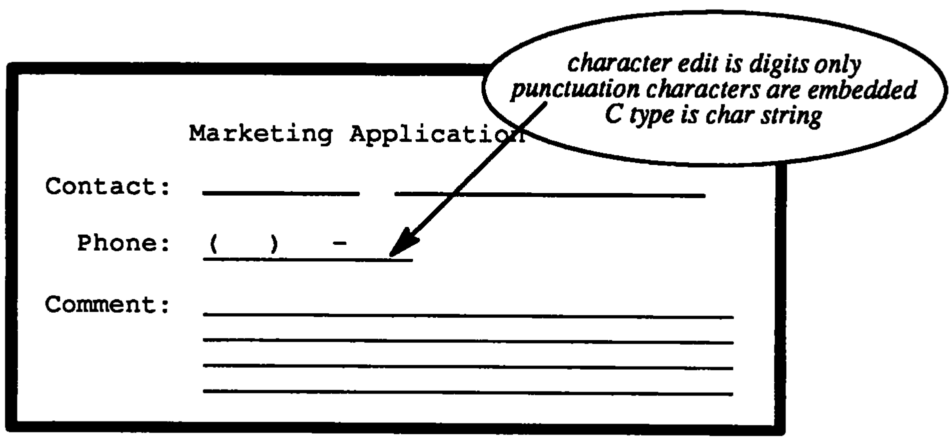



Figure 38:

JAM/DB*i* uses character edits to determine a JAM type if the field or LDB variable does **not** have any of the following edits: date/time, currency, or data type (excluding `omit` and `char string`).

| Character Filter | Format Type |
|---|---|
| `digits only` | `ft_unsigned` |
| `yes/no field` | `dt_yesno` |
| `numeric (+, -, .)` | `ft_double` |
| all other | `ft_char` |

22.1.3.

## Field Edits

Developers may also use field edits to enforce some integrity checks at the application front-end. Remember that field edits are not enforced until the field is validated.

The field edits right justified and null field are enforced when **JAM**/DB*i* writes SELECT data to a field.

By default, **JAM** distinguishes between empty fields and null fields. To make **JAM** and **JAM**/DB*i* treat a blank field as null, you must modify the message file:

```
SM_NULLEDIT = " "
```

22.1.4.

## Field Attachments

The following field attachments are useful in a **JAM**/DB*i* application:

- field name
- item selection
- table lookup

We discuss them below.

### Field Name

This is the only required field characteristic for a **JAM**/DB*i* field. Database values cannot be written to unnamed fields.

Usually the developer gives a field the same name as a database column. The case of the field name is very important. In the `vendor_list` structure in `dbiinit.c` the developer sets a case flag for the engine. If the flag is `DM_FORCE_TO_LOWER_CASE` the developer must use lower case for the database fields. If the flag is `DM_FORCE_TO_UPPER_CASE` the developer must use upper case for the database fields. If the flag is DM_PRESERVE_CASE the developer must use the exact case of the column names for the database fields.

A developer may also alias a database column to a JAM variable. This is done with the command DBMS ALIAS. When aliasing is used, the developer may use any valid JAM variable.

### Item Selection and Table Lookup Screens

These attachments often improve an application's user interface. The screen entry function of the lookup or selection screen may query the database for lookup or selection values. Since the application saves the query, rather than the values, the screen maintains itself.

Developers may use the JAM library function sm_svscreen to save the selection or lookup screen in memory at runtime. If the screen is saved in memory, the application will not need to execute the query each time it displays the lookup or selection screen.

See the JAM Author's Guide and Programmer's Guide for more information.

## 22.1.5. Miscellaneous Edits

Developers may execute database functions from any of the field hook functions attached in this window. Two of the miscellaneous edits may be used to format data, the date time edit and the currency edit.

### JAM TYPE:

| Miscellaneous Edit | Format Type | Precision |
|---|---|---|
| date or time field | DT_DATETIME | n/a |
| currency format | DT_CURRENCY | from places edit |

If data is fetched to fields with either of these edits, the database values are automatically formatted with the date-time or currency edit.

## 22.1.6. Field Size

The length of a field should generally be the same as the width of its associated database column. If the column is very wide, set field length to a smaller size and set the maximum shifting length to the column width. If a field's maximum length is not equal to the width of its associated column, surplus data is truncated without warning.

Developers should set the number of elements and occurrences for a JAM/DB*i* field according to the screen size and the type of query. If a query is designed to return only one row at a time, developers should create a field with one element for each column in the row. If the query is designed to return multiple rows, the developers should create an array for each column in the row. Developers may create a scrolling array by setting the maximum number of occurrences to the greatest number of rows that may be retrieved. Developers may also create a non-scrolling array.

In brief, the two approaches are:

- Retrieve all qualifying records into large *scrolling arrays*. Each array represents a database column. The arrays usually have the same number of occurrences, so that array occurrences with the same occurrence number represent a database row. Developers may use `@dmrowcount` to ensure that the number of rows selected is less than the number of array occurrences. Users scroll through the arrays with the PgUp and PgDn keys (logical keys `SPGU` and `SPGD`). Developers may also install a customized scroll driver for an array. See the JAM Programmer's Guide for details.
- Retrieve *n* qualifying records incrementally into *non-scrolling arrays*. In MS-DOS environments where memory is limited, developers may wish to limit the number of rows read in at any one time. For each column, developers create an array with *n* non-scrolling occurrences. The first select retrieves the first *n* rows. Each subsequent DBMS CONTINUE retrieves the next *n* rows. To make this arrangement invisible to the user, the developers may use a key change function or a keyset to map the DBMS CONTINUE call to the user's physical PgDn key. Of course, the function may also be called by a standard function key. To support backward scrolling, the application may use a continuation file. A continuation file is created with the DBMS STORE command.

Developers may use the word-wrap edit to write long character strings to an array.

22.1.7.

## Data Type

JAM data type edits have no affect on the application interface. In other words, JAM does not validate a field's contents against its data type edit and developer's cannot use this feature to perform frontend integrity checks. Developer's may use it however to set a field's format type.

When determining a variable's format type, JAM/DB*i* first checks the data type edit. If a C type is explicitly set, the keyword `TYPE` will appear on the field's summary window (`PF5` in draw mode of the **JAM** Screen Editor). If there is no explicit data type, or it is `omit` JAM/DB*i* will examine the variable's date-time, currency, and character edits to determine a format type. The data type edits which set format types are listed below.

| Data Type | Format Type | Precision |
|---|---|---|
| `char string` | `FT_CHAR` | |
| `int` | `FT_INT` | |
| `unsigned int` | `FT_UNSIGNED` | |
| `short int` | `FT_SHORT` | |
| `long int` | `FT_LONG` | |
| `float` | `FT_FLOAT` | `yes` |
| `double` | `FT_DOUBLE` | `yes` |
| `zoned dec.` | `FT_ZONED` | |
| `packed dec.` | `FT_PACKED` | |

# Symbols


:: *Overview* 27; *Developers* 72—76

:+ *Overview* 32; *Developers* 62—68

:= *Developers* 68—69

@ *Developers* 93; *Reference* 113—114


# A


Aggregate functions: *Developers* 81—82

Aliases: *Overview* 10; *Developers* 79—82

Autocommit. *See* Transaction

AVG. *See* Aggregate functions


# B


Binary columns: *Reference* 131, 181

Binding: *Overview* 27; *Developers* 72—76
  examples: *Developers* 74


# C


Case sensitivity: *Overview* 20; *Developers* 53
  alias names: *Developers* 80
  connection names: *Developers* 55
  cursor names: *Developers* 57
  engine names: *Developers* 52
  field names: *Developers* 79
  keywords: *Appendices* 1

Colon preprocessing: *Overview* 32, 33, 36; *Developers* 62—71
  colon equal: *Developers* 68
  colon plus: *Developers* 62—68
  examples: *Developers* 69—71
  simulating from C: *Reference* 181

Commit
  *See also* Transaction

Connection: *Developers* 55—56; *Reference* 129—130
  closing: *Developers* 55, 56, 60
  current: *Developers* 56
  declaring: *Developers* 55
  declaring, options. *See* Engine specific Notes
  default: *Developers* 55, 56
  using more than one: *Overview* 42; *Developers* 55, 60

Continuation File: *Developers* 85

Currency edits: *Developers* 67, 89—90

Cursor: *Overview* 27, 36; *Developers* 57; *Reference* 130
  declaring: *Developers* 58
  default: *Developers* 57
  executing: *Developers* 59
  executing with parameters: *Developers* 59
  maximum number of. *See* Engine specific Notes
  named: *Developers* 58
  redeclaring: *Developers* 60


# D


Data dictionary: *Overview* 44

Date and time edit: *Developers* 66, 89

dbiinit.c: *Overview* 5, 7, 19—21; *Developers* 52 dbms: *Developers* 48—49

DBMS commands: *Reference* 129—132
  ALIAS: *Developers* 79—82
  CATQUERY: *Developers* 92
  COMMIT: *Developers* 104
  CONTINUE: *Developers* 85—88
  CONTINUE_BOTTOM: *Developers* 86—88
  CONTINUE_TOP: *Developers* 86
  CONTINUE_UP: *Developers* 86
  FORMAT: *Developers* 92
  OCCUR: *Developers* 88
  ONENTRY: *Developers* 96—97
  ONERROR: *Developers* 99—102
  ONEXIT: *Developers* 98—99
  ROLLBACK: *Developers* 104
  START: *Developers* 88
  STORE FILE: *Developers* 85—88
  UNIQUE: *Developers* 90—91

DBMS functions, ROLLBACK: *Developers* 105—107

dbms versus sql: *Developers* 48

dm_
  @dm variables: *Reference* 113—114
  dm_ library functions: *Reference* 181—182

## E

Engine: *Overview* 3, 7, 42; *Developers* 52; *Reference* 129
  accessing: *Developers* 55
  current: *Developers* 54, 56
  de-installing: *Reference* 181
  default: *Developers* 54, 56
  errors: *Developers* 93
  initializing: *Overview* 19; *Developers* 52, 54; *Reference* 181
  using more than one: *Overview* 41; *Reference* 179

Errors: *Overview* 11, 38, 39; *Reference* 131
  @dmengerrcode, @dmengerrmsg: *Reference* 113, 115—116, 117
  @dmretcode, @dmretmsg: *Reference* 113, 122—123, 124
  customized processing: *Overview* 38, 44; *Developers* 98—102
  default processing: *Developers* 94
  displaying error messages: *Overview* 38; *Developers* 53
  engine-specific error codes: *Developers* 93; *Reference* 115—116, 117
  error handler: *Overview* 38; *Developers* 98—102
    sample: *Overview* 24
  generic DBi error codes: *Developers* 93; *Reference* 122, 124; *Appendices* 1—3
  transactions: *Developers* 105—107
  warning codes: *Developers* 93

## F

f2tbl: *Reference* 206—212

Field characteristics, affecting formatting and colon preprocessing: *Developers* 64—66

Formatting text for a database: *Developers* 62—71, 73—76

Formatting text from a database: *Developers* 89

## G

Global error and status variables: *Reference* 113

## H

Hook functions: *Developers* 95; *Reference* 131

## I

Identifier
  case sensitivity for field names: *Developers* 79
  column name: *Developers* 79

Initialization: *Overview* 19
  engines: *Reference* 181
  JAM/DBi: *Reference* 181

## J

JAM type: *Developers* 63, 64, 89
  C type: *Developers* 66, 71
  character: *Developers* 67, 69—70, 89
  currency: *Developers* 66, 67—71
  date and time: *Developers* 66, 70, 89
  null: *Developers* 64, 70
  numeric: *Developers* 67—71, 89

JPL versus C: *Developers* 49

## L

Logging on and off. *See* Connection

## M

MAX. *See* Aggregate functions

Multiple rows, retrieving: *Overview* 37; *Developers* 83—88

## N

No more rows status: *Developers* 84; *Reference* 122, 124

Null: *Developers* 64, 68—69

Number of rows fetched: *Developers* 83—88
  @dmrowcount: *Reference* 114, 125

## P

Parameters: *Developers* 72—76
  binding: *Developers* 73

Precision: *Developers* 89—90

## R

Rollback. *See* Transaction

## S

SELECT: *Developers* 78—82; *Reference* 130
  aliasing: *Overview* 10; *Developers* 79—82
  automatic mapping: *Overview* 10; *Developers* 79
  binary columns: *Reference* 131
  concatenating result row: *Developers* 92
  destination of: *Overview* 10; *Developers* 78—82, 92
    aggregate functions: *Developers* 81
  format of results: *Developers* 89
  no more rows: *Developers* 84; *Reference* 125
  number of rows fetched: *Developers* 83—88; *Reference* 125

scrolling: *Developers* 83—88; *Reference* 130
suppressing repeating values: *Developers* 90—91
unique column values: *Developers* 90—91
writing to a file: *Developers* 92
writing to a specific occurrence: *Developers* 83, 88
writing to word-wrapped arrays: *Developers* 83

Serial
*See also* Engine specific Notes
@dmserial: *Reference* 114, 127—128

sql: *Developers* 48

SQL syntax: *Overview* 40; *Developers* 47

Stored procedure
*See also* Engine specific Notes
return code, @dmengreturn: *Reference* 114, 118—119

SUM. *See* Aggregate functions

Support routine: *Overview* 3, 7, 19, 20, 41; *Developers* 52

# T

tbl2f: *Reference* 213—225

Transaction: *Overview* 41; *Developers* 103—107
*See also* Engine specific Notes
error handling: *Developers* 105—107

# U

Utilities: *Reference* 205

# V

Variables, global @dm: *Reference* 113

# W

Warnings, @dmengwarncode, @dmengwarnmsg: *Reference* 113—114, 120, 121

WITH clause: *Reference* 175, 177, 179

Word wrapped edit: *Developers* 83