ORACLE
JD EDWARDS ENTERPRISEONE

JD Edwards EnterpriseOne Tools 8.96 Developments Standards for Business Function Programming Guide

April 2006

ORACLE

JD Edwards EnterpriseOne Tools 8.96 Developments Standards for Business Function Programming Guide
SKU E1_TOOLS896TDS-B 0406

Contents

About This Documentation Preface

JD Edwards EnterpriseOne implementation guides provide you with the information that you need to implement and use JD Edwards EnterpriseOne applications from Oracle.

This preface discusses:

- JD Edwards EnterpriseOne application prerequisites.
- Application fundamentals.
- Documentation updates and printed documentation.
- Additional resources.
- Typographical conventions and visual cues.
- Comments and suggestions.
- Common fields in implementation guides.

Note. Implementation guides document only elements, such as fields and check boxes, that require additional explanation. If an element is not documented with the process or task in which it is used, then either it requires no additional explanation or it is documented with common fields for the section, chapter, implementation guide, or product line. Fields that are common to all JD Edwards EnterpriseOne applications are defined in this preface.

JD Edwards EnterpriseOne Application Prerequisites

To benefit fully from the information that is covered in these books, you should have a basic understanding of how to use JD Edwards EnterpriseOne applications.

You might also want to complete at least one introductory training course, if applicable.

You should be familiar with navigating the system and adding, updating, and deleting information by using JD Edwards EnterpriseOne menus, forms, or windows. You should also be comfortable using the World Wide Web and the Microsoft Windows or Windows NT graphical user interface.

These books do not review navigation and other basics. They present the information that you need to use the system and implement your JD Edwards EnterpriseOne applications most effectively.

Application Fundamentals

Each application implementation guide provides implementation and processing information for your JD Edwards EnterpriseOne applications.

For some applications, additional, essential information describing the setup and design of your system appears in a companion volume of documentation called the application fundamentals implementation guide. Most product lines have a version of the application fundamentals implementation guide. The preface of each implementation guide identifies the application fundamentals implementation guides that are associated with that implementation guide.

The application fundamentals implementation guide consists of important topics that apply to many or all JD Edwards EnterpriseOne applications. Whether you are implementing a single application, some combination of applications within the product line, or the entire product line, you should be familiar with the contents of the appropriate application fundamentals implementation guides. They provide the starting points for fundamental implementation tasks.

Documentation Updates and Printed Documentation

This section discusses how to:

- Obtain documentation updates.
- Order printed documentation.

Obtaining Documentation Updates

You can find updates and additional documentation for this release, as well as previous releases, on Oracle's PeopleSoft Customer Connection website. Through the Documentation section of Oracle's PeopleSoft Customer Connection, you can download files to add to your Implementation Guides Library. You'll find a variety of useful and timely materials, including updates to the full line of JD Edwards EnterpriseOne documentation that is delivered on your implementation guides CD-ROM.

Important! Before you upgrade, you must check Oracle's PeopleSoft Customer Connection for updates to the upgrade instructions. Oracle continually posts updates as the upgrade process is refined.

See Also

Oracle's PeopleSoft Customer Connection, http://www.oracle.com/support/support_peoplesoft.html

Ordering Printed Documentation

You can order printed, bound volumes of the complete line of JD Edwards EnterpriseOne documentation that is delivered on your implementation guide CD-ROM. Oracle makes printed documentation available for each major release of JD Edwards EnterpriseOne shortly after the software is shipped. Customers and partners can order this printed documentation by using any of these methods:

- Web
- Telephone
- Email

Web

From the Documentation section of Oracle's PeopleSoft Customer Connection website, access the PeopleBooks Press website under the Ordering PeopleBooks topic. Use a credit card, money order, cashier's check, or purchase order to place your order.

Telephone

Contact MMA Partners, the book print vendor, at 877 588 2525.

Email

Send email to MMA Partners at peoplebookspress@mmapartner.com.

See Also

Oracle's PeopleSoft Customer Connection, http://www.oracle.com/support/support_peoplesoft.html

Additional Resources

The following resources are located on Oracle's PeopleSoft Customer Connection website:

Resource	Navigation
Application maintenance information	Updates + Fixes
Business process diagrams	Support, Documentation, Business Process Maps
Interactive Services Repository	Support, Documentation, Interactive Services Repository
Hardware and software requirements	Implement, Optimize, and Upgrade; Implementation Guide; Implementation Documentation and Software; Hardware and Software Requirements
Installation guides	Implement, Optimize, and Upgrade; Implementation Guide; Implementation Documentation and Software; Installation Guides and Notes
Integration information	Implement, Optimize, and Upgrade; Implementation Guide; Implementation Documentation and Software; Pre-Built Integrations for PeopleSoft Enterprise and JD Edwards EnterpriseOne Applications
Minimum technical requirements (MTRs) (JD Edwards EnterpriseOne only)	Implement, Optimize, and Upgrade; Implementation Guide; Supported Platforms
Documentation updates	Support, Documentation, Documentation Updates
Implementation guides support policy	Support, Support Policy
Prerelease notes	Support, Documentation, Documentation Updates, Category, Release Notes
Product release roadmap	Support, Roadmaps + Schedules
Release notes	Support, Documentation, Documentation Updates, Category, Release Notes
Release value proposition	Support, Documentation, Documentation Updates, Category, Release Value Proposition
Statement of direction	Support, Documentation, Documentation Updates, Category, Statement of Direction

Resource	Navigation
Troubleshooting information	Support, Troubleshooting
Upgrade documentation	Support, Documentation, Upgrade Documentation and Scripts

Typographical Conventions and Visual Cues

This section discusses:

- Typographical conventions.
- Visual cues.
- Country, region, and industry identifiers.
- Currency codes.

Typographical Conventions

This table contains the typographical conventions that are used in implementation guides:

Typographical Convention or Visual Cue	Description
Bold	Indicates PeopleCode function names, business function names, event names, system function names, method names, language constructs, and PeopleCode reserved words that must be included literally in the function call.
Italics	Indicates field values, emphasis, and JD Edwards EnterpriseOne or other book-length publication titles. In PeopleCode syntax, italic items are placeholders for arguments that your program must supply. We also use italics when we refer to words as words or letters as letters, as in the following: Enter the letter *O*.
KEY+KEY	Indicates a key combination action. For example, a plus sign (+) between keys means that you must hold down the first key while you press the second key. For ALT+W, hold down the ALT key while you press the W key.
`Monospace font`	Indicates a PeopleCode program or other code example.
“ ” (quotation marks)	Indicate chapter titles in cross-references and words that are used differently from their intended meanings.

Typographical Convention or Visual Cue	Description
. . . (ellipses)	Indicate that the preceding item or series can be repeated any number of times in PeopleCode syntax.
{ } (curly braces)	Indicate a choice between two options in PeopleCode syntax. Options are separated by a pipe (|).
[ ] (square brackets)	Indicate optional items in PeopleCode syntax.
& (ampersand)	When placed before a parameter in PeopleCode syntax, an ampersand indicates that the parameter is an already instantiated object. Ampersands also precede all PeopleCode variables.

Visual Cues

Implementation guides contain the following visual cues.

Notes

Notes indicate information that you should pay particular attention to as you work with the JD Edwards EnterpriseOne system.

Note. Example of a note.

If the note is preceded by *Important!,* the note is crucial and includes information that concerns what you must do for the system to function properly.

Important! Example of an important note.

Warnings

Warnings indicate crucial configuration considerations. Pay close attention to warning messages.

Warning! Example of a warning.

Cross-References

Implementation guides provide cross-references either under the heading "See Also" or on a separate line preceded by the word *See.* Cross-references lead to other documentation that is pertinent to the immediately preceding documentation.

Country, Region, and Industry Identifiers

Information that applies only to a specific country, region, or industry is preceded by a standard identifier in parentheses. This identifier typically appears at the beginning of a section heading, but it may also appear at the beginning of a note or other text.

Example of a country-specific heading: “(FRA) Hiring an Employee”

Example of a region-specific heading: “(Latin America) Setting Up Depreciation”

Country Identifiers

Countries are identified with the International Organization for Standardization (ISO) country code.

Region Identifiers

Regions are identified by the region name. The following region identifiers may appear in implementation guides:

- Asia Pacific
- Europe
- Latin America
- North America

Industry Identifiers

Industries are identified by the industry name or by an abbreviation for that industry. The following industry identifiers may appear in implementation guides:

- USF (U.S. Federal)
- E&G (Education and Government)

Currency Codes

Monetary amounts are identified by the ISO currency code.

Comments and Suggestions

Your comments are important to us. We encourage you to tell us what you like, or what you would like to see changed about implementation guides and other Oracle reference and training materials. Please send your suggestions to Documentation Manager, Oracle Corporation, 7604 Technology Way, Denver, CO, 80237. Or email us at documentation_us@oracle.com.

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions.

Common Fields Used in Implementation Guides

Address Book Number Enter a unique number that identifies the master record for the entity. An address book number can be the identifier for a customer, supplier, company, employee, applicant, participant, tenant, location, and so on. Depending on the application, the field on the form might refer to the address book number as the customer number, supplier number, or company number, employee or applicant ID, participant number, and so on.

As If Currency Code Enter the three-character code to specify the currency that you want to use to view transaction amounts. This code enables you to view the transaction amounts as if they were entered in the specified currency rather than the foreign or domestic currency that was used when the transaction was originally entered.

Batch Number Displays a number that identifies a group of transactions to be processed by the system. On entry forms, you can assign the batch number or the system can assign it through the Next Numbers program (P0002).

Batch Date Enter the date in which a batch is created. If you leave this field blank, the system supplies the system date as the batch date.

Batch Status Displays a code from user-defined code (UDC) table 98/IC that indicates the posting status of a batch. Values are:

Blank: Batch is unposted and pending approval.

A: The batch is approved for posting, has no errors and is in balance, but has not yet been posted.

D: The batch posted successfully.

E: The batch is in error. You must correct the batch before it can post.

P: The system is in the process of posting the batch. The batch is unavailable until the posting process is complete. If errors occur during the post, the batch status changes to *E*.

U: The batch is temporarily unavailable because someone is working with it, or the batch appears to be in use because a power failure occurred while the batch was open.

Branch/Plant Enter a code that identifies a separate entity as a warehouse location, job, project, work center, branch, or plant in which distribution and manufacturing activities occur. In some systems, this is called a business unit.

Business Unit Enter the alphanumeric code that identifies a separate entity within a business for which you want to track costs. In some systems, this is called a branch/plant.

Category Code Enter the code that represents a specific category code. Category codes are user-defined codes that you customize to handle the tracking and reporting requirements of your organization.

Company Enter a code that identifies a specific organization, fund, or other reporting entity. The company code must already exist in the F0010 table and must identify a reporting entity that has a complete balance sheet.

Currency Code Enter the three-character code that represents the currency of the transaction. JD Edwards EnterpriseOne provides currency codes that are recognized by the International Organization for Standardization (ISO). The system stores currency codes in the F0013 table.

Document Company Enter the company number associated with the document. This number, used in conjunction with the document number, document type, and general ledger date, uniquely identifies an original document.

If you assign next numbers by company and fiscal year, the system uses the document company to retrieve the correct next number for that company.

If two or more original documents have the same document number and document type, you can use the document company to display the document that you want.

Document Number — Displays a number that identifies the original document, which can be a voucher, invoice, journal entry, or time sheet, and so on. On entry forms, you can assign the original document number or the system can assign it through the Next Numbers program.

Document Type — Enter the two-character UDC, from UDC table 00/DT, that identifies the origin and purpose of the transaction, such as a voucher, invoice, journal entry, or time sheet. JD Edwards EnterpriseOne reserves these prefixes for the document types indicated:

P: Accounts payable documents.

R: Accounts receivable documents.

T: Time and pay documents.

I: Inventory documents.

O: Purchase order documents.

S: Sales order documents.

Effective Date — Enter the date on which an address, item, transaction, or record becomes active. The meaning of this field differs, depending on the program. For example, the effective date can represent any of these dates:

- The date on which a change of address becomes effective.
- The date on which a lease becomes effective.
- The date on which a price becomes effective.
- The date on which the currency exchange rate becomes effective.
- The date on which a tax rate becomes effective.

Fiscal Period and **Fiscal Year** — Enter a number that identifies the general ledger period and year. For many programs, you can leave these fields blank to use the current fiscal period and year defined in the Company Names & Number program (P0010).

G/L Date (general ledger date) — Enter the date that identifies the financial period to which a transaction will be posted. The system compares the date that you enter on the transaction to the fiscal date pattern assigned to the company to retrieve the appropriate fiscal period number and year, as well as to perform date validations.

JD Edwards EnterpriseOne Tools Development Standards for Business Function Programming Preface

This preface discusses Development Standards for Business Function Programming companion documentation.

Development Standards for Business Function Programming Companion Documentation

Additional, essential information describing the setup and design of Oracle's JD Edwards EnterpriseOne Tools resides in companion documentation. The companion documentation consists of important topics that apply to Business Function Programming as well as other JD Edwards EnterpriseOne Tools. You should be familiar with the contents of these companion guides:

- Development Tools: APIs and Business Functions
- Development Tools: Data Dictionary
- Development Tools: Event Rules and System Functions

See Also

JD Edwards EnterpriseOne Tools 8.96 Development Tools: APIs and Business Functions Guide, "Getting Started with JD Edwards EnterpriseOne Tools: APIs and Business Functions"

JD Edwards EnterpriseOne Tools 8.96 Development Tools: Data Dictionary Guide, "Getting Started with JD Edwards EnterpriseOne Data Dictionary"

JD Edwards EnterpriseOne Tools 8.96 Development Tools: Event Rules Guide, "Getting Started with JD Edwards EnterpriseOne Tools Development Tools: Event Rules"

CHAPTER 1

Getting Started with JD Edwards EnterpriseOne Tools Development Standards for Business Function Programming

This chapter discusses:

- Development Standards for Business Function Programming Overview.
- Development Standards for Business Function Programming Implementation.

Development Standards for Business Function Programming Overview

Business Function Programming is an integral part of Oracle's JD Edwards EnterpriseOne tool set. Application developers can attach custom functionality to application and batch processing events by using business functions. You program business functions are programmed in C code, discussed in this guide, or as Named Event Rules.

Development Standards for Business Function Programming Implementation

This section provides an overview of the steps that are required to implement Development Standards for Business Function Programming.

In the planning phase of your implementation, take advantage of all JD Edwards EnterpriseOne sources of information, including the installation guides and troubleshooting information. A complete list of these resources appears in the preface in *About This Documentation* with information about where to find the most current version of each.

Business Function Programming Implementation Steps

This table lists the steps for JD Edwards EnterpriseOne Tools Business Function Programming implementation.

Step	Reference
1. Set up default project in OMW.	*JD Edwards EnterpriseOne Tools 8.96 Object Management Workbench Guide*, "Configuring JD Edwards EnterpriseOne OMW," Understanding JD Edwards EnterpriseOne OMW Configuration
2. Configure OMW transfer activity rules and allowed actions.	*JD Edwards EnterpriseOne Tools 8.96 Object Management Workbench Guide*, "Configuring JD Edwards EnterpriseOne OMW," Understanding JD Edwards EnterpriseOne OMW Configuration
3. Set up default location and printers.	*JD Edwards EnterpriseOne Tools 8.96 Development Tools: Report Printing Administration Technologies Guide*, "Getting Started with JD Edwards EnterpriseOne Report Printing Administration Technologies"

CHAPTER 2

Understanding Naming Conventions

This chapter discusses:

- Source and header file names.
- Function names.
- Variable names.
- Business function (BSFN) data structure names.

Source and Header File Names

Source and header file names can be a maximum of 8 characters and should be formatted as *bxxyyyy*, where:

- *b* = BSFN object
- *xx* (second two digits) = The system code, such as:
  - 01 = Address Book
  - 04 = Accounts Payable
- *yyyyy* (the last five digits) = A sequential number for the system code, such as:
  - 00001 = The first source or header file for the system code
  - 00002 = The second source or header file for the system code

Both the C source and the accompanying header file should have the same name.

This table shows examples of this naming convention:

System	System Code	Source Number	Source File	Header File
Address Book	01	10	b0100010.c	b0100010.h
Accounts Receivable	04	58	b0400058.c	b0400058.h
General Ledger	09	2457	b0902457.c	b0902457.h

Function Names

An internal function can be a maximum of 42 characters and should be formatted as *Ixxxxxx_a*, where:

- *I* = An internal function
- *xxxxxx* = The source file name
- *a* = The function description

  Function descriptions can be up to 32 characters in length, and must not contain spaces. Be as descriptive as possible and capitalize the first letter of each word, such as ValidateTransactionCurrencyCode. When possible use the major table name or purpose of the function.

An example of a Function Name is I4100040_CompareDate

Note. Do not use an underscore after I.

Variable Names

Variables are storage places in a program and can contain numbers and strings. Variables are stored in the computer's memory. Variables are used with keywords and functions, such as **char** and **MATH_NUMERIC**, and must be declared at the beginning of the program.

A variable name can be up to 32 characters in length. Be as descriptive as possible and capitalize the first letter of each word.

You must use Hungarian prefix notation for all variable names, as shown in this table:

Prefix	Description
c	JCHAR
sz	NULL-terminated JCHAR string
z	ZCHAR
zz	NULL-terminated ZCHAR string
n	short
l	long
b	Boolean
mn	MATH_NUMERIC
jd	JDEDATE
lp	long pointer
i	integer
by	byte
ul	unsigned long (identifier)
us	unsigned Short

Prefix	Description
ds	data structures
h	handle
e	enumerated types
id	id long integer, JDE data type for returns
ut	JDEUTIME
sz	VARCHAR

Example: Hungarian Notation for Variable Names

These variable names use Hungarian notation:

JCHAR	cPaymentRecieved;
JCHAR [ ]	szCompanyNumber = _J(00000);
short	nLoopCounter;
long int	lTaxConstant;
BOOL	bIsDateValid;
MATH_NUMERIC	mnAddressNumber;
JDEDATE	jdGLDate;
LPMATH_NUMERIC	lpAddressNumber;
int	iCounter;
byte	byOffsetValue;
unsigned long	ulFunctionStatus;
D0500575A	dsInputParameters;
JDEDB_RESULT	idJDEDBResult;

Business Function Data Structure Names

The data structure for business function event rules and business functions should be formatted as *DxxyyyyA*, where:

- *D* = Data structure

- *xx* (second two digits) = The system code, such as
  - 01 = Address Book
  - 02 = Accounts Payable
- *yyyy* = A next number (the numbering assignments follow current procedures in the respective application groups)
- *A* = An alphabetical character (such as A, B, C and so on) placed at the end of the data structure name to indicate that a function has multiple data structures

  Even if a function has only one data structure, you should include the A in the name.

An example of a Business Function Data Structure Name is D050575A.

CHAPTER 3

Ensuring Readability

This chapter provides an overview of readability and discusses how to:

- Maintain the source and header code change log.
- Indent code.
- Format compound statements.

Understanding Readability

Readable code is easier to debug and maintain. You can make code more readable by maintaining the change log, inserting comments, indenting code, and formatting compound statements.

Maintaining the Source and Header Code Change Log

You must note any code changes that you make to the standard source and header for a business function. Include this information:

- *SAR* - the SAR number
- *Date* - the date of the change
- *Initials* - the programmer's initials
- *Comment* - the reason for the change

Inserting Comments

Insert comments that describe the purpose of the business function and your intended approach. Using comments will make future maintenance and enhancement of the function easier.

Use this checklist for inserting comments:

- Always use the /*comment */ style. The use of // comments is not portable.
- Precede and align comments with the statements they describe.
- Comments should never be more that 80 characters wide.

Example: Inserting Comments

This example shows the correct way to insert block and inline comments into code:

```
/*-----------------------------------------------------------
 * Comment blocks need to have separating lines between
 * the text description. The separator can be a
 * dash '-' or an asterisk '*'
 *-----------------------------------------------------------*/
 if ( statement )
 {
    statements
 } /* inline comments indicate the meaning of one statement */
 /*-----------------------------------------------------------
 * Comments should be used in all segments of the source
 * code. The original programmer may not be the programmer
 * maintaining the code in the future which makes this a
 * crucial step in the development process.
 *-----------------------------------------------------------*/
 /*************************************************************
 * Function Clean Up
 *************************************************************/
```

Indenting Code

Any statements executed inside a block of code should be indented within that block of code. Standard indentation is three spaces.

Note. Set up the environment for the editor you are using to set tab stops at 3 and turn the tab character display off. Then, each time you press the Tab key, three spaces are inserted rather than the tab character. Select auto-indentation.

Example: Indenting Code

This the standard method to indent code:

```
function block
{
   if ( nJDEDBReturn == JDEDB_PASSED )
   {
      CallSomeFunction( nParameter1, szParameter2 );
      CallAnotherFunction( lSomeNumber );
      while( FunctionWithBooleanReturn() )
      {
         CallYetAnotherFunction( cStatusCode );
      }
   }
}
```


Formatting Compound Statements

Compound statements are statements followed by one or more statements enclosed with braces. A function block is an obvious example of a compound statement. Control statements (while, for) and selection statements (if, switch) are also examples of compound statements.

Omitting braces is a common C coding practice when only one statement follows a control or selection statement. However, you must use braces for all compound statements for these reasons:

- The absence of braces can cause errors.
- Braces ensure that all compound statements are treated the same way.
- In the case of nested compound statements, the use of braces clarifies the statements that belong to a particular code block.
- Braces make subsequent modifications easier.

Refer to these guidelines when formatting compound statements:

- Always have one statement per line within a compound statement.
- Always use braces to contain the statements that follow a control statement or a selection statement.
- Braces should be aligned with the initial control or selection statement.
- Logical expressions evaluated within a control or selection statement should be broken up across multiple lines if they do not fit on one line. When breaking up multiple logical expressions, do not begin a new line with the logical operator; the logical operator must remain on the preceding line.
- When evaluating multiple logical expressions, use parentheses to explicitly indicate precedence.
- Never declare variables within a compound statement, except function blocks.
- Use braces for all compound statements.
- Place each opening or closing brace, { or }, on a separate line.

Example: Formatting Compound Statements

This example shows how to format compound statements for ease of use and to prevent mistakes:

```
/*
 * Do the Issues Edit Line if the process edits is either
 * blank or set to SKIP_COMPLETIONS. The process edits is
 * set to SKIP_COMPLETIONS if Hours and Quantities is in
 * interactive mode and Completions is Blind in P31123.
 */
if ((dsWorkCache.PO_cIssuesBlindExecution == _J('1')) &&
   ((dsCache.cPayPointCode == _J('M'))      ||
    (dsCache.cPayPointCode == _J('B')))       &&
   (lpDS->cProcessEdits != ONLY_COMPLETIONS))
{
   /* Process the Pay Point line for Material Issues */
   idReturnCode = I3101060_BlindIssuesEditLine(&dsInternal,
                                               &dsCache,
                                               &dsWorkCache);
}
```

Example: Using Braces to Clarify Flow

This example shows the use of braces to clarify the flow and prevent mistakes:

```
if(idJDBReturn != JDEDB_PASSED)
{
   /* If not add mode, record must exist */
   if ((lpdsInternal->cActionCode != ADD_MODE) &&
       (lpdsInternal->cActionCode != ATTACH_MODE))
   {
      /* Issue Error 0002 - Work Order number invalid */
      jdeStrncpy((JCHAR*)(lpdsInternal->szErrorMessageID),
                 (const JCHAR*)_J(0002),
                 DIM(lpdsInternal->szErrorMessageID)-1);
      lpdsInternal->idFieldID = IDERRmnOrderNumber_15;
      idReturnCode = ER_ERROR;
   }
}
else
{
   /* If in add mode and the record exists, issue error and exit */
   if (lpdsInternal->cActionCode == ADD_MODE)
   {
      /* Issue Error 0002 - Work Order number invalid */
      jdeStrncpy((JCHAR*)(lpdsInternal->szErrorMessageID),
                 (const JCHAR*)_J(0002),
                 DIM(lpdsInternal->szErrorMessageID)-1);
      lpdsInternal->idFieldID = IDERRmnOrderNumber_15;
      idReturnCode = ER_ERROR;
   }
   else
   {
      /*
       * Set flag used in determining if the F4801 record should be sent
       * in to the modules
       */
       lpdsInternal->cF4801Retrieved = _J('1');
   }
}
```

Example: Using Braces for Ease in Subsequent Modifications

The use of braces prevents mistakes when the code is later modified. Consider this example. The original code contains a test to see if the number of lines is less than a predefined limit. As intended, the return value is assigned a certain value if the number of lines is greater than the maximum. Later, someone decides that an error message should be issued in addition to assigning a certain return value. The intent is for both statements to be executed only if the number of lines is greater than the maximum. Instead, **idReturn** will be set to **ER_ERROR** regardless of the value of **nLines**. If braces were used originally, this mistake would have been avoided.

```
ORIGINAL
```



```
if (nLines > MAX_LINES)
   idReturn = ER_ERROR;

MODIFIED

if (nLines > MAX_LINES)
   jdeErrorSet (lpBhvrCom, lpVoid,
               (ID) 0, _J(4353), (LPVOID) NULL);
   idReturn = ER_ERROR;

STANDARD ORIGINAL

if (nLines > MAX_LINES)
{
   idReturn = ER_ERROR;
}

STANDARD MODIFIED

if (nLines > MAX_LINES)
{
   jdeErrorSet (lpBhvrCom, lpVoid,
               (ID) 0, _J(4363), (LPVOID) NULL);
   idReturn = ER_ERROR;
}
```

Example: Handling Multiple Logical Expressions

This example shows how to handle multiple logical expressions:

```
while ( (lWorkArray[elWorkX] < lWorkArray[elWorkMAX]) &&
        (lWorkArray[elWorkX] < lWorkArray[elWorkCDAYS]) &&
        (idReturnCode == ER_SUCCESS))


{
   statements
}
```

CHAPTER 4

Declaring and Initializing Variables and Data Structures

This chapter provides an overview of variables and data structures, and discusses how to:

- Use define statements
- Use typedef statements
- Create function prototypes
- Initialize variables
- Initialize data structures
- Use standard variables

Understanding Variables and Data Structures

Variables and data structures must be defined and initialized before they can be used to store data. This chapter describes how to declare and initialize them. It includes topics on using define statements, using **typedef**, creating function prototypes, initializing variables, initializing data structures, and using standard variables.

Using Define Statements

A define statement is a directive that sets up constants at the beginning of the program. A define statement always begins with a pound sign (#). All business functions include the system header file: jde.h. System-wide define statements are included in the system header file.

If you need define statements for a specific function, include the define statement in uppercase letters within the source file for the function whenever possible. The statement should directly follow the header file inclusion statement.

Usually, you should place define statements in the source file, not the header file. When placed in the header file, you can redefine the same constant with different values, causing unexpected results. However, rare cases exist when it is necessary to place a define statement in the function header file. In these cases, precede the definition name with the business function name to ensure uniqueness.

Example: #define in Source File

This example includes define statements within a business function source file:

```
/*************************************************************
* Notes
*************************************************************/


#include <bxxxxxxx.h>


/*************************************************************
 * Global Definitions
 *************************************************************/
#define CACHE_GET            '1'
#define CACHE_ADD            '2'
#define CACHE_UPDATE         '3'
#define CACHE_DELETE         '4'
```

Example: #define in Header File

This example includes define statements within a business function header:

```
/*********************************************************
 * External Business Function Header Inclusions
 *********************************************************/

#include <bxxxxxxx.h>

/*********************************************************
 * Global definitions
 *********************************************************/
#define BXXXXXXX_CACHE_GET         '1'
#define BXXXXXXX_CACHE_ADD         '2'
#define BXXXXXXX_CACHE_UPDATE      '3'
#define BXXXXXXX_CACHE_DELETE      '4'
```

Using Typedef Statements

When using **typedef** statements, always name the object of the **typedef** statement using a descriptive, uppercase format. If you are using a **typedef** statement for data structures, remember to include the name of the business function in the name of the **typedef** to make it unique. See the example for using a **typedef** statement for a data structure.

Example: Using Typedef for a User-Defined Data Structure

This is an example of a user-defined data structure:

```
/*************************************************************
 * Structure Definitions
 *************************************************************/
```



```
typedef struct
{
   HUSER         hUser;           /** User handle **/
   HREQUEST      hRequestF0901;   /** File Pointer to the
                                    * Account Master **/
   DSD0051       dsData;          /** X0051 - F0902 Retrieval **/
   int           iFromYear;       /** Internal Variables **/
   BOOL          bProcessed;
   MATH_NUMERIC  mnCalculatedAmount;
   JCHAR         szSummaryJob[13];
   JDEDATE       jdStartPeriodDate;
} DSX51013_INFO, *LPDSX51013_INFO;
```

Creating Function Prototypes

Refer to these guidelines when defining function prototypes:

- Always place function prototypes in the header file of the business function in the appropriate prototype section.
- Include function definitions in the source file of the business function, preceded by a function header.
- Ensure that function names follow the naming convention defined in this guide.
- Ensure that variable names in the parameter list follow the naming convention defined in this guide.
- List the variable names of the parameters along with the data types in the function prototype.
- List one parameter per line so that the parameters are aligned in a single column.
- Do not allow the parameter list to extend beyond 80 characters in the function definition. If the parameter list must be broken up, the data type and variable name must stay together. Align multiple-line parameter lists with the first parameter.
- Include a return type for every function. If a function does not return a value, use the keyword **void** as the return type.
- Use the keyword **void** in place of the parameter list if nothing is passed to the function.

Example: Creating a Business Function Prototype

This is an example of a standard business function prototype:

```
/*************************************************************
 * Business Function: BusinessFunctionName
 *
 *    Description: Business Function Name
 *
 *     Parameters:
 *      LPBHVRCOM  lpBhvrCom Business Function Communications
 *      LPVOID   lpVoid  Void Parameter - DO NOT USE!
 *    LPDSD51013 lpDS   Parameter Data Structure Pointer
 *
```

```
 ****************************************************************/

JDEBFRTN (ID) JDEBFWINAPI BusinessFunctionName (LPBHVRCOM  lpBhvrCom,
                                                LPVOID     lpVoid,
                                                LPDSXXXXXX lpDS)
```

Example: Creating an Internal Function Prototype

This is an example of a standard internal function prototype:

```
Type XXXXXXXX_AAAAAAAA( parameter list ... );

type     : Function return value
XXXXXXXX : Unique source file name
AAAAAAAA : Function Name
```

Example: Creating an External Business Function Definition

This is an example of a standard external business function definition:

```
/*
 *  see sample source for standard business function heading
 */
JDEBFRTN (ID) JDEBFWINAPI GetAddressBookDescription(LPBHVRCOM  lpBhvrCom,
                                                    LPVOID     lpVoid,
                                                    LPDSNNNNNN lpDS)
{
   ID idReturn = ER_SUCCESS;
   /*---------------------------------------------------------------
    * business function code
    */
   return idReturn;
}
```

Example: Creating an Internal Function Definition

This is an example of a standard internal function definition:

```
/*--------------------------------------------------------------------------
 *  see sample source for standard function header
 */
void I4100040_GetSupervisorManagerDefault( LPBHVRCOM lpBhvrCom,
                                           LPSTR lpszCostCenterIn,
                                           LPSTR lpszManagerOut,
                                           LPSTR lpszSupervisorOut )
/*--------------------------------------------------------------------------
 * Note: b4100040 is the source file name
 */
{
   /*
    * internal function code
    */
}
```


Initializing Variables

Variables store information in memory that is used by the program. Variables can store strings of text and numbers.

When you declare a variable, you should also initialize it. Two types of variable initialization exist: explicit and implicit. Variables are explicitly initialized if they are assigned a value in the declaration statement. Implicit initialization occurs when variables are assigned a value during processing.

This information covers standards for declaring and initializing variables in business functions and includes an example of standard formats.

Use these guidelines when declaring and initializing variables:

- Declare variables using this format:

```
datatype variable name = initial value; /* descriptive comment*/
```

- Declare all variables used within business functions and internal functions at the beginning of the function. Although C allows you to declare variables within compound statement blocks, this standard requires all variables used within a function to be declared at the beginning of the function block.
- Declare only one variable per line, even if multiple variables of the same type exist. Indent each line three spaces and left align the data type of each declaration with all other variable declarations. Align the first character of each variable name (**variable name** in the preceding format example) with variable names in all other declarations.
- Use the naming conventions set forth in this guide. When initializing variables, the initial value is optional depending on the data type of the variable. Generally, all variables should be explicitly initialized in their declaration.
- The descriptive comment is optional. In most cases, variable names are descriptive enough to indicate the use of the variable. However, provide a comment if further description is appropriate or if an initial value is unusual.
- Left align all comments.
- Data structures should be initialized to zero using the **memset** function immediately after the declaration section.
- Some Application Program Interfaces (APIs), such as the JDB ODBC API, provide initialization routines. In this case, the variables intended for use with the API should be initialized with the API routines.
- Always initialize pointers to NULL and include an appropriate type call at the declaration line.
- Initialize all variables, except data structures, in the declaration.
- Initialize all declared data structures, **MATH_NUMERIC**, and **JDEDATE** to NULL.
- Ensure that the byte size of the variable matches the size of the data structure you want to store.

Example: Initializing Variables

This example shows how to initialize variables:

```
JDEBFRTN (ID) JDEBFWINAPI F0902GLDateSensitiveRetrieval
                          (LPBHVRCOM   lpBhvrCom,
                           LPVOID     lpVoid,
                           LPDSD0051   lpDS)
/************************************************************
```

```
 * Variable declarations
 ************************************************************/
   ID             idReturn      = ER_SUCCESS;
   JDEDB_RESULT   eJDEDBResult  = JDEDB_PASSED;
   long           lDateDiff     = 0L;
   BOOL           bAddF0911Flag = TRUE;
   MATH_NUMERIC   mnPeriod      = {0};

/************************************************************
 * Declare structures
 ************************************************************/
   HUSER          hUser         = (HUSER) NULL;
   HREQUEST       hRequestF0901 = (HREQUEST) NULL;
   DSD5100016     dsDate        = {0};
   JDEDATE        jdMidDate     = {0};

/************************************************************
 * Pointers
 ************************************************************/
   LPX0051_DSTABLES lpdsTables = (LPX0051_DSTABLES) 0L;

/************************************************************
 * Check for NULL pointers
 ************************************************************/
   if ((lpBhvrCom == (LPBHVRCOM) NULL) ||
       (lpVoid    == (LPVOID)    NULL) ||
       (lpDS      == (LPDSD0051)   NULL))
   {
      jdeErrorSet (lpBhvrCom, lpVoid, (ID) 0,
                   _J(4363), (LPVOID) NULL);
      return ER_ERROR;
   }

/************************************************************
 * Main Processing
 ************************************************************/
   eJDEDBResult = JDB_InitBhvr ((void*)lpBhvrCom,
                                &hUser,
                                (JCHAR *) NULL,
                                JDEDB_COMMIT_AUTO);

 memcopy ((void*)) &dsDate.jdPeriodEndDate,
          (const void*) &lpDS->jdGLDate, sizeof(JDEDATE));
```

Initializing Data Structures

When writing to the table, the table recognizes these default values:

- Space-NULL if string is blank
- 0 value if math numeric is 0
- 0 **JDEDATE** if date is blank
- Space if character is blank

Always **memset** to NULL on the data structure that is passed to another business function to update a table or fetch a table.

Example: Using Memset to Reset the Data Structure to Null

This example resets the data structure to NULL when initializing the data structure:

```
bOpenTable = B5100001_F5108SetUp( lpBhvrCom, lpVoid,
                                  lphUser, &hRequestF5108);

if ( bOpenTable )
{
   memset( (void *)(&dsF5108Key), 0x00, sizeof(KEY1_F5108) );
   jdeStrcpy( (JCHAR*) dsF5108Key.mdmcu,
              (const JCHAR*) lpDS->szBusinessUnit );
   memset( (void *)(&dsF5108), 0x00, sizeof(F5108) );

   jdeStrcpy( (JCHAR*) dsF5108.mdmcu,
            (const JCHAR*) lpDS->szBusinessUnit );
   MathCopy(&dsF5108.mdbsct, &mnCentury);
   MathCopy(&dsF5108.mdbsfy, &mnYear);
   MathCopy(&dsF5108.mdbtct, &mnCentury);
   MathCopy(&dsF5108.mdbtfy, &mnYear);
   eJDEDBResult = JDB_InsertTable( hRequestF5108,
                                   ID_F5108,
                                   (ID)(0),
                                   (void *) (&dsF5108) );
}
```

Using Standard Variables

This section discusses how to:

- Use flag variables.
- Use input and output parameters.
- Use fetch variables.

Using Flag Variables

When creating flag variables, use these guidelines:

- Any true-or-false flag used must be a Boolean type (**BOOL**).

- Name the flag variable to answer a question of TRUE or FALSE.

These are examples of flag variables, with a brief description of how each is used:

Flag Variable	Description
bIsMemoryAllocated	Apply to memory allocation
bIsLinkListEmpty	Link List

Using Input and Output Parameters

Business functions frequently return error codes and pointers. The input and output parameters in the business function data structure should be named as follows:

Input and Output Parameter	Description
cReturnPointer	When allocating memory and returning **GENLNG**.
cErrorCode	Based on cCallType, cErrorCode returns a 1 when it fails or a 0 when it succeeds.
cSuppressErrorMessage	If the value is 1, do not display error message using **jdeErrorSet(...)**. If the value is 0, display the error.
szErrorMessageId	If an error occurs, return an error message ID (value). Otherwise, return four spaces.

Using Fetch Variables

Use fetch variables to retrieve and return specific information, such as a result; to define the table ID; and to specify the number of keys to use in a fetch.

Fetch Variable	Description
idJDEDBResult	APIs or JD Edwards EnterpriseOne functions, such as **JDEDB_RESULT**
idReturnValue	Business function return value, such as **ER_WARNING** or **ER_ERROR**
idTableXXXXID	Where XXXX is the table name, such as F4101 and F41021, the variable used to define the Table ID.
idIndexXXXXID	Where XXXX is the table name, such as F4101 or F41021, the variable used to define the Index ID of a table.
usXXXXNumColToFetch	Where XXXX is the table name, such as F4101 and F41021, the number of the column to fetch. *Do not* put the literal value in the API functions as the parameter.
usXXXXNumOfKeys	Where XXXX is the table name, such as F4101 and F41021, the number of keys to use in the fetch.

Example: Using Standard Variables

This example illustrates the use of standard variables:

```
/*************************************************************
 * Variable declarations
 *************************************************************/
ID      idJDEDBResult  = JDEDB_PASSED;
ID      idTableF0901   = ID_F0901;
ID      idIndexF0901   = ID_F0901_ACCOUNT_ID;
ID      idFetchCol[]   = { ID_CO, ID_AID, ID_MCU, ID_OBJ,
                           ID_SUB, ID_LDA, ID_CCT };
ushort    usNumColToFetch = 7;
ushort    usNumOfKeys     = 1;

/*************************************************************
 * Structure declarations
 *************************************************************/
KEY3_F0901     dsF0901Key = {0}
DSX51013_F0901 dsF0901 = {0}

/*************************************************************
 * Main Processing
 *************************************************************/
/** Open the table, if it is not open **/
  if ((*lpdsInfo->lphRequestF0901) == (HREQUEST) NULL)
  {
     if ( (*lpdsInfo->lphUser) == (HUSER) 0L )
     {
        idJDEDBResult = JDB_InitBhvr ((void*)lpBhvrCom,
                                      &lpdsInfo->lphUser,
                                      (JCHAR *) NULL,
                                      JDEDB_COMMIT_AUTO);
     }
     if (idJDEDBResult == JDEDB_PASSED)
     {
        idJDEDBResult = JDB_OpenTable( (*lpdsInfo->lphUser),
                                        idTableF0901,
                                        idIndexF0901,
                                        (LPID)(idFetchCol),
                                        (ushort)(usNumColFetch),
                                        (JCHAR *) NULL,
                                        &lpdsInfo->hRequestF0901 );
     }
  }
/** Retrieve Account Master - AID only sent **/
if (idJDEDBResult == JDEDB_PASSED)
{
   /** Set Key and Fetch Record **/
   memset( (void *)(&dsF0901Key),
```

```
            (int) _J('\0'), sizeof(KEY3_F0901) );
   jdeStrcpy ((char *) dsF0901Key.gmaid,
              (const JCHAR*) lpDS->szAccountID );
   idJDEDBResult = JDB_FetchKeyed ( lpdsInfo->hRequestF0901,
                                    idIndexF0901,
                                    (void *)(&dsF0901Key),
                                    (short)(1),
                                    (void *)(&dsF0901),
                                    (int)(FALSE) );
   /** Check for F0901 Record **/
   if (eJDEDBResult == JDEDB_PASSED)
   {
      statement
   }
}
```


CHAPTER 5

Applying General Coding Guidelines

This chapter discusses how to:

- Use function calls.
- Pass pointers between business functions.
- Allocate and release memory.
- Use **hRequest** and **hUser**.
- Typecast.
- Comparison test.
- Copy strings with **jdeStrcpy** or **jdeStrncpy**.
- Use the function clean up area.
- Insert function exit points.
- Terminate a function.

Using Function Calls

Reuse of existing functions through a function call prevents duplicate code. Refer to these guidelines when using function calls:

- Always put a comma between each parameter. Optionally, you can add a space for readability.
- If the function has a return value, always check the return of the function for errors or a valid value.
- Use **jdeCallObject** to call another business function.
- When calling functions with long parameter lists, the function call should not be wider than 80 characters.

  Break the parameter list into one or more lines, aligning the first parameter of proceeding lines with the first parameter in the parameter list.
- Make sure the data types of the parameters match the function prototype.

  When intentionally passing variables with data types that do not match the prototype, explicitly cast the parameters to the correct data type.

Calling an External Business Function

Use **jdeCallObject** to call an external business function defined in the Object Management Workbench. Include the header file for the external business function that contains the prototype and data structure definition. It is good practice to check the value of the return code.

Example: Calling an External Business Function

This example calls an external business function:

```
/*------------------------------------------------------------
 *
 * Retrieve account master information
 *
 *------------------------------------------------------------*/
   idReturnCode = jdeCallObject(_J("ValidateAccountNumber),
                                NULL,
                                lpBhvrCom,
                                lpVoid,
                                (void*) &dsValidateAccount,
                                (CALLMAP*) NULL,
                                (int) 0,
                                (JCHAR*) NULL,
                                (JCHAR*) NULL,
                                (int) 0 );
   if ( idReturnCode == ER_SUCCESS )
   {
      statement;
   }
```

Calling an Internal Business Function

You can access internal business functions (internal C functions) within the same source file.

You may create modular subroutines that can be accessed from multiple source files. Use `CALLIBF(fcn(parm1,parm2))` and`CALLIBFRET(ret,fcn(parm1,parm2))` to access internal business functions within a different source file but within the same DLL. Use **CALLIBF** to call an internal business function with no return value. Use **CALLIBFRET** to call an internal business function with a return value. Both **CALLIBF** and **CALLIFBRET** can call internal business functions with any type or number of parameters.

CALLIBF and **CALLIBFRET** can only call internal functions within the same business function DLL. They cannot call functions in other business function DLLs. For example, if the internal function **intFcn123()** is in B550001.C, which is in the CALLBSFN.DLL, you cannot called it with **CALLIBF** or **CALLIBFRET** from a business function in CDIST.DLL.

To use **CALLIBF** or **CALLIBFRET** for an internal business function, the business function must have its prototype in the business function header. If you do not want other modules calling the internal business function, place the prototype in the C file, not the header file.

Calling internal business functions has several advantages over external business functions. First, they do not have the **jdeCallObject** performance overhead of checking OCM mapping and possibly executing the function remotely. A called function always executes in the same process from where it was called. Second, the parameters are not restricted to JD Edwards EnterpriseOne data dictionary data types. Any valid C data type, including pointers, may be passed in and out of internal functions.

Example: Calling an Internal Business Function with No Return Value

This example calls an internal business function that has no return value.

This portion is an example of b550001.h:


```
/* normal business function header pieces */
...
/* The internal business function prototype must be in the header for other
   modules to call it */
   void i550001(int *a, int b);
```

This portion is an example of b550001.c:

```
/* normal business function code pieces */
#include <b550001.h>
JDEBFRTN(ID) JDEBFWINAPI TestBSFN(LPBHVRCOM  lpVhvrCom,
                                  LPVOID   lpVoid,
                                  LPDSB550001 lpDS)
{
...
}
void i550001(int *a, int b)
{
   *a = *a + b;
   return;
}
```

This portion is an example of b550002.c:

```
/* normal business function code pieces */
#include <b550002.h>
#include <b550001.h>

JDEBFRTN(ID) JDEBFWINAPI TestBSFN(LPBHVRCOM  lpBhvrCom,
                                  LPVOID   lpVoid,
                                  LPDSB550001 lpDS)
{
   int total = 3;
   int adder = 7;

   CALLIBF(i550001(&total,adder));
}
```

Example: Calling an Internal Business Function with a Return Value

This example calls an internal business function that has a return value.

This portion is an example of b550001.h:

```
/* normal business function header pieces */
...
/* The internal business function prototype must be in the header for
other modules to call it */

   int i550001(int a, int b);
```

This portion is an example of b550001.c:

```
/* normal business function code pieces */
#include <b550001.h>
```

```
JDEBFRTN(ID) JDEBFWINAPI TestBSFN(LPBHVRCOM   lpBhvrCom,
                                  LPVOID      lpVoid,
                                  LPDSB550001 lpDS)
{
...
}
int i550001(int a, int b)
{
   a = a + b;
   return;
}
```

This portion is an example of b550002.c:

```
/* normal business function code pieces */
#include <b550002.h>
#include <b550001.h>

JDEBFRTN(ID) JDEBFWINAPI TestBSFN(LPBHVRCOM   lpBhvrCom,
                                  LPVOID      lpVoid,
                                  LPDSB550001 lpDS)
{
   int total  = 0;
   int adder1 = 6;
   int adder2 = 7;
   CALLIBFRET(total,i550001(adder1,adder2));
}
```

Passing Pointers between Business Functions

Never pass pointers directly in or out of business functions. A pointer memory address should not be greater than 32 bits. If you pass a pointer address that exceeds 32 bits across the platform to a client that supports just 32 bits, the significant digit might be truncated and invalidate the address.

The correct way to share pointers between business functions is to store the address in an array. This array is located on the server platform specified in the Object Configuration Manager (OCM). The array allows up to 100 memory locations to be allocated and stored, and it is maintained by JD Edwards EnterpriseOne tools. The index to a position in the array is a long integer type or ID. Use the **GENLNG** data dictionary object in the business function data structure to pass this index in or out of the business function.

Storing an Address in an Array

Use **jdeStoreDataPtr** to store an allocated memory pointer in an array for later retrieval. The index to the position in the array is returned. This index should be passed out through the business function data structure (**lpDS**).

Example: Storing an Address in an Array

This example illustrates how to store an address in an array:

```
If (lpDS->cReturnF4301PtrFlag == _J('1'))
```



```
{
   lpDS->idF4301RowPtr = jdeStoreDataPtr(hUser,(void *)lpdsF4301);
}
```

Retrieving an Address from an Array

Use **jdeRetrieveDataPtr** to retrieve an address outside the current business function. The index to the position in the array should be passed in through the business function data structure (**lpDS**). When you use **jdeRetrieveDataPtr**, the address remains in the array and can be retrieved again later.

Example: Retrieving an Address from an Array

This example retrieves an address from an array:

```
lpdsF43199 = (LPF43199) jdeRetrieveDataPtr
              (hUser, lpDS->idF43199Pointer);
```

Removing an Address from an Array

Use **jdeRemoveDataPtr** to remove the address from the array cell and release the array cell. The index to the position in the array should be passed in through the business function data structure (**lpDS**). A corresponding call to **jdeRemoveDataPtr** must exist for every **jdeStoreDataPtr**. If you use **jdeAlloc** to allocate memory, use **jdeFree** to free the memory.

Example: Removing an Address from an Array

This example removes an address from an array:

```
if (lpDS->idGenericLong != (ID) 0)
{
   lpGenericPtr = (void *)jdeRemoveDataPtr(hUser,lpDS->idGenericLong);
   if (lpGenericPtr != (void *) NULL)
   {
      jdeFree((void *)lpGenericPtr);
      lpDS->idGenericLong = (ID) 0;
      lpGenericPtr = (void *) NULL;
   }
}
```

Allocating and Releasing Memory

Use **jdeAlloc** to allocate memory. Because **jdeAlloc** affects performance, use it sparingly.

Use **jdeFree** to release memory within a business function. For every **jdeAlloc**, a **jdeFree** should exist to release the memory.

Note. Use the business function FreePtrToDataStructure, B4000640, to release memory through event rule logic.

Example: Allocating and Releasing Memory within a Business Function

This example uses **jdeAlloc** to allocate memory, and then, in the function cleanup section, **jdeFree** to release memory:

```
statement
lpdsF4301 = (LPF4301)jdeAlloc( COMMON_POOL,sizeof(F4301),MEM_ZEROINIT ) ;
statement

/************************************************************
 * Function Clean Up Section
 ************************************************************/
if (lpdsF4301 != (LPF4301) NULL)
{
   jdeFree( lpdsF4301 );
}
```

Using hRequest and hUser

Some API calls require either an**hUser** or an**hRequest** variable, or both. To get the **hUser**, use **JDBInitBhvr**. To get the **hRequest**, use **JDBOpenTable**. Initialize **hUser** and **hRequest** to NULL in the variable declaration line. All **hRequest** and **hUser** declarations should have **JDB_CloseTable( )** and **JDB_FreeBhvr( )** in the function cleanup section.

Typecasting

Typecasting is also known as type conversion. Use typecasting when the function requires a certain type of value, when defining function parameters, and when allocating memory with **jdeAlloc()**.

Note. This standard is for all function calls as well as function prototypes.

Comparison Testing

Always use explicit tests for comparisons. Do not embed assignments in comparison tests. Assign a value or result to a variable and use the variable in the comparison test.

Always test floating point variables using <= or >=. Do not use == or != since some floating point numbers cannot be represented exactly.

Example: Comparison Test

This example shows how to create C code for comparison tests.

```
eJDEDBResult = JDB_InitBhvr ((void*)lpBhvrCom,
                             &hUser,
                             (JCHAR *) NULL,
                             JDEDB_COMMIT_AUTO);
```



```
/** Check for Valid hUser **/
if (eJDEDBResult == JDEDB_PASSED)
{
   statement;
}
```

Example: Creating TRUE or FALSE Test Comparison that Uses Boolean Logic

This example is a TRUE or FALSE test comparison that uses Boolean logic:

```
/* IsStringBlank has a BOOL return type. It will always return either
 * TRUE or FALSE */
if ( IsStringBlank( szString) )
{
   statement;
}
```

Copying Strings with jdeStrcpy or jdeStrncpy

When copying strings of the same length, such as business unit, you may use the **jdeStrcpy** ANSI API. If the strings differ in length-as with a description-use the **jdeStrncpy** ANSI API with the number of characters you need returned, not counting the trailing NULL character.

```
/*********************************************************
 * Variable Definitions
 *********************************************************/
 JCHAR      szToBusinessUnit(13);
 JCHAR      szFromBusinessUnit(13);
 JCHAR      szToDescription(31);
 JCHAR      szFromDescription(41);
/*********************************************************
 * Main Processing
 *********************************************************/
   jdeStrcpy((JCHAR *) szToBusinessUnit,
             (const JCHAR *) szFromBusinessUnit );


   jdeStrncpy((JCHAR *) szToDescription,
              (const JCHAR *) szFromDescription,
              DIM(szToDescription)-1 );
```

Using the Function Clean Up Area

Use the function clean up area to release any allocated memory, including **hRequest** and **hUser**.

Example: Using the Function Clean Up Area to Release Memory

This example shows how to release memory in the function clean up area:

```
lpdsF4301 = (LPF4301)jdeAlloc( COMMON_POOL,
                               sizeof(F4301),MEM_ZEROINIT ) ;
/************************************************************
 * Function Clean Up Section
 ************************************************************/
if (lpdsF4301 != (LPF4301 ) NULL)
{
   jdeFree( lpdsF4301 );
}

if (hRequestF4301 != (HREQUEST) NULL)
{
   JDB_CloseTable( hRequestF4301 );
}

JDB_FreeBhvr( hUser );

return ( idReturnValue ) ;
```

Inserting Function Exit Points

Where possible, use a single exit point (return) from the function. The code is more structured when a business function has a single exit point. The use of a single exit point also enables the programmer to perform cleanup, such as freeing memory and terminating ODBC requests, immediately before the return. In more complex functions, this action might be difficult or unreasonable. Include the necessary cleanup logic, such as freeing memory and terminating ODBC requests, when programming an exit point in the middle of a function.

Use the return value of the function to control statement execution. Business functions can have one of two return values: **ER_SUCCESS** or **ER_ERROR**. By initializing the return value for the function to **ER_SUCCESS**, the return value can be used to determine the processing flow.

Example: Inserting an Exit Point in a Function

This example illustrates the use of a return value for the function to control statement execution:

```
  ID         idReturn        = ER_SUCCESS;
/************************************************************
 * Main Processing
 ************************************************************/
   memset( (void *)(&dsInfo), 0x00, sizeof(DSX51013_INFO) );
   idReturn = X51013_VerifyAndRetrieveInformation( lpBhvrCom,
                                                   lpVoid,
                                                   lpDS,
                                                   &dsInfo );
 /** Check for Errors and Company or Job Level Projections **/
 if ( (idReturn == ER_SUCCESS) &&
```



```
      (lpDS->cJobCostProjections == _J('Y')) )
 {
    /** Process All Periods between the From and Thru Dates **/
    while ( (!dsInfo.bProcessed) &&
            (idReturn == ER_SUCCESS) )
    {
        /** Retrieve Calculation Information **/
        if ((dsInfo.bRetrieveBalance) && (idReturn == ER_SUCCESS))
        {
           idReturn = X51013_RetrieveAccountBalances( lpBhvrCom,
                                                      lpVoid,
                                                      lpDS,
                                                      &dsInfo );
        }
        if (idReturn == ER_SUCCESS)
        {
           statement;
        }
    } /* End Processing */
 }

/*****************************************************************
 * Function Clean Up
 *****************************************************************/
   if ( (dsInfo.hUser) != (HUSER) NULL )
   {
      statement;
   }

   return idReturn;
```

Terminating a Function

Always return a value with the termination of a function.

CHAPTER 6

Coding for Portability

This chapter provides overviews of portability concepts and portability guidelines, and discusses how to avoid common server build errors and warnings.

Portability Concepts

Portability is the ability to run a program on more than one system platform without modifying it. JD Edwards EnterpriseOne is a portable environment. This chapter presents considerations and guidelines for porting objects between systems.

Standards that affect the development of relational database systems are determined by:

- ANSI (American National Standards Institute) standard
- X/OPEN (European body) standard
- ISO SQL standard

Ideally, industry standards enable users to work identically with different relational database systems. Each major vendor supports industry standards but also offers extensions to enhance the functionality of the SQL language. In addition, vendors constantly release upgrades and new versions of their products.

These extensions and upgrades affect portability. Due to the effect of software development on the industry, applications need a standard interface to databases-an interface that will not be affected by differences among database vendors. When vendors provide a new release, the effect on existing applications needs to be minimal. To solve portability issues, many organizations have moved to standard database interfaces, called open database connectivity (ODBC).

Portability Guidelines

Refer to these guidelines to develop business functions that comply with portability standards:

- Business functions must be ANSI-compatible for portability.

  Since different computer platforms might present limitations, exceptions to this rule do exist. However, do not use a non-ANSI exception without approval from the Business Function Standards Committee.

- Do not create a program that depends on data alignment, because each system aligns data differently by allocating bytes or words.

  For example: for a one-character field that is one byte. Some systems allocate only one byte for that field, while other systems allocate the entire word for the field.

- Keep in mind that vendor libraries and function calls are system-dependent and exclusive to that vendor.

  This means that if the program is compiled using a different compiler, that particular function will fail.
- Use caution when using pointer arithmetic because it is system-dependent and is based on the data alignment.
- Do not assume that all systems will initialize a variable the same way.

  Always explicitly initialize variables.
- Use caution when using an offset to explicitly retrieve a value within the data structure.

  This guideline also relates to data alignment. Use offset to define cache index.
- Always typecast if your parameter does not match the function parameter.

Note. JCHAR szArray[13] is not the same as **(JCHAR *)** in the function declaration. Therefore, typecast of **(JCHAR *)** is required for **szArray** for that particular function.

- Never typecast on the left-hand side of the assignment statement, as it can result in a loss of data.

  For example, in the statement `(short)nValue = (long),lValue` will lose the value of the long integer if it is too large to fit into a short integer data type.
- Do not use C++ comments (C++ comments begin with two forward slashes).

Preventing Common Server Build Errors and Warnings

JD Edwards EnterpriseOne business functions must be ANSI-compatible for portability. Since different computer platforms and servers have their own limitations, our business functions must comply with all server standards. This topic presents guidelines for coding business functions that correctly build on different servers.

Comments within Comments

Never use comments that are included in other comments. Each "/*" should be followed by subsequent "*/". Refer to these examples.

Example: C Comments that Comply with the ANSI Standard

Use this C standard comment block:

```
/*********************************************************************
* Correct Method of C Comments             *
*********************************************************************/
/* SAR 1234567 Begin*/
/* Populate the lpDS->OrderedPlacedBy value from the userID only in
   the ADD mode */
   if ( lpDS->cHeaderActionCode == _J('1'))
   {
      if (IsStringBlank(lpDS->szOrderedPlacedBy))
     .{
         jdeStrcpy((JCHAR *)(lpDS->szOrderedPlacedBy),
                   (const JCHAR *)(lpDS->szUserID));
      }/* End of defaulting in the user id into Order placed by
```



```
            if the later was left blank */
   }/* SAR 1234567 End */
```

Example: C Comments that Comply with the ANSI Standard

Use this C standard comment block:

```
/*************************************************************************
* Correct Method of C Comments             *
*************************************************************************/
/* SAR 1234567 Begin*/
/* Populate the lpDS->OrderedPlacedBy value from the userID only in
   the ADD mode */
   if ( lpDS->cHeaderActionCode == _J('1'))
   {
      if (IsStringBlank(lpDS->szOrderedPlacedBy))
     .{
         jdeStrcpy((JCHAR *)(lpDS->szOrderedPlacedBy),
                   (const JCHAR *)(lpDS->szUserID));
      }/* End of defaulting in the user id into Order placed by
            if the later was left blank */
   }/* SAR 1234567 End */
```

Example: Comments within Comments Cause Problems on Different Servers

This example shows that comments within comments can cause problem on different servers:

```
/*************************************************************************
 C Comments within Comments Causing Server Build Errors and Warnings
 *************************************************************************/
/* SAR 1234567 Begin
/* Populate the lpDS->OrderedPlacedBy value from the userID only in
   the ADD mode */
*/
   if ( lpDS->cHeaderActionCode == _J('1'))
   {
      if (IsStringBlank(lpDS->szOrderedPlacedBy))
      {
         jdeStrcpy((JCHAR *)(lpDS->szOrderedPlacedBy),
                   (const JCHAR *)(lpDS->szUserID));
      }/* End of defaulting in the user id into the Order placed by
       /* if the later was left blank */
   }/* SAR 1234567 End */
```

New Line Character at the End of a Business Function

Some servers need a new line character at the end of the source and header file in order to build correctly. It is a best practice to ensure that a new line character is added at the end of each business function. Press the Enter key at the end of the code to add a new line character.

Use of Null Character

Be careful when using NULL character '\0'. This character starts with a back slash. Using '/0' is an error that is not reported by the compiler.

Example: Use of NULL Character

This example shows an incorrect and a correct use of the NULL character:

```
/*************Initialize Data Structures**************************/
/*Error Code*/
/* '/0' is used assuming it to be a NULL character*/
/* memset((void *)(&dsVerifyActivityRulesStatusCodeParms),
          (int)('/0'), sizeof(DSD4000260A));*/


/*Correct Use of NULL Character*/
memset((void *)(&dsVerifyActivityRulesStatusCodeParms),
       (int)('\0'), sizeof(DSD4000260A));
```

Lowercase Letters in Include Statements

When an external business function or table is included in the header file, use lowercase letters in the include statement. Uppercase letters cause build errors.

Example: Use of Lowercase Letters in Include Statements

This example shows the incorrect and correct use of lowercase letters in the include statement:

```
/*********************************************************************
* External Business Function Header Inclusions
*********************************************************************/
/*Incorrect method of including external business function header*/
/*Include Statement Causing Build Warnings on Various Servers*/
   #include <B0000130.h>
/*Correct method of including external business function header*/
   #include <b0000130.h>
```

Initialized Variables that are Not Referenced

Each variable that is declared and initialized under the Variables Declaration section in the business function must be used in the program. For example: if the variable idReturnValue is initialized, then it must be used somewhere in the program.

CHAPTER 7

Understanding JD Edwards EnterpriseOne Defined Structures

Oracle's JD Edwards EnterpriseOne provides two data types that should concern you when you create business functions: **MATH_NUMERIC** and **JDEDATE**. Since these data types might change, use the Common Library APIs provided by JD Edwards EnterpriseOne to manipulate them. Do not access the members of these data types directly.

This chapter discusses:

- The **MATH_NUMERIC** data type.
- The **JDEDATE** data type.

MATH_NUMERIC Data Type

The **MATH_NUMERIC** data type is commonly used to represent numeric values in JD Edwards EnterpriseOne software. This data type is defined as follows:

```
struct tag MATH_NUMERIC

{
   ZCHAR String [MAXLEN_MATH_NUMERIC + 1];
   BYTE  Sign;
   ZCHAR EditCode;
   short nDecimalPosition;
   short nLength;
   WORD  wFlags;
   ZCHAR szCurrency [4];
   Short nCurrencyDecimals;
   short nPrecision;
};

typedef struct tag MATH_NUMERIC MATH_NUMERIC, FAR *LPMATH_NUMERIC;
```

This table shows math-numeric elements and their descriptions:

MATH_NUMERIC Element	Description
String	The digits without separators
Sign	A minus sign indicates the number is negative. Otherwise, the value is 0x00.

MATH_NUMERIC Element	Description
EditCode	The data dictionary edit code used to format the number for display
nDecimalPosition	The number of digits from the right to place the decimal
nLength	The number of digits in the String
wFlags	Processing flags
szCurrency	Currency code
nCurrencyDecimals	The number of currency decimals
nPrecision	The data dictionary size

When assigning **MATH_NUMERIC** variables, use the MathCopy API. MathCopy copies the information, including Currency, into the location of the pointer. This API prevents any lost data in the assignment.

Initialize local **MATH_NUMERIC** variables with the ZeroMathNumeric API. If a **MATH_NUMERIC** is not initialized, invalid information, especially currency information, might be in the data structure, which can result in unexpected results at runtime.

```
/************************************************
 * Variable Definitions
 ************************************************/
   MATH_NUMERIC   mnVariable  = {0};

/************************************************
 * Main Processing
 ************************************************/
   ZeroMathNumeric( &mnVariable );
   MathCopy( &mnVariable,
             &lpDS->mnVariable );
```

JDEDATE Data Type

The **JDEDATE** data type is commonly used to represent dates in JD Edwards EnterpriseOne. The data type is defined as follows:

```
struct tag JDEDATE
{
   short nYear;
   short nMonth;
   short nDay;
};

typedef struct tag JDEDATE JDEDATE, FAR *LPJDEDATE;
```


JDEDATE Element	Description
nYear	The year
nMonth	The month
nDay	The day

Using Memcpy to Assign JDEDATE Variables

When assigning **JDEDATE** variables, use the **memcpy** function. The **memcpy** function copies the information into the location of the pointer. If you use a flat assignment, you might lose the scope of the local variable in the assignment, which could result in a lost data assignment.

```
/*************************************************
 * Variable Definitions
 ************************************************/
   JDEDATE   jdToDate;
/*************************************************
 * Main Processing
 ************************************************/
   memcpy((void*) &jdToDate,
          (const void *) &lpDS->jdFromDate,
          sizeof(JDEDATE) );
```

JDEDATECopy

You can use **JDEDATECopy**, as well as **memcpy**, to assign **JDEDATE** variables. The syntax is as follows:

```
#define JDEDATECopy(pDest, pSource)
        memcpy(pDest, pSource, sizeof(JDEDATE))
```


CHAPTER 8

Implementing Error Messages

This chapter provides and overview of error messages and discusses how to: ok

- Insert parameters for error messages in **lpDS**.
- Initialize behavior errors.
- Use text substitution to display specific error messages.
- Map data structure errors.

Understanding Error Messages

Messages provide an effective and usable method of communicating information to end-users. You can use simple messages or text substitution messages.

Text substitution messages provide specific information to the user. At runtime, the system replaces variables in the message with substitution values. Two types of text substitution messages exist:

- Error messages (glossary group E)
- Workflow messages (glossary group Y)

The return code from all JDB and JDE Cache APIs must be checked and an appropriate error message set, returned, or both to the calling function. The standard error messages for JDB and JDE Cache errors are shown in these tables.

The JDB errors are:

Error ID	Description
078D	Open Table Failed
078E	Close Table Failed
078F	Insert to Table Failed
078G	Delete from Table Failed
078H	Update to Table Failed
078I	Fetch from Table Failed
078J	Select from Table Failed

Error ID	Description
078K	Set Sequence of Table Failed
078S *	Initialization of Behavior Failed

* 078S does not use text substitution

The JDE Cache errors are:

Error ID	Description
078L	Initialization of Cache Failed
078M	Open Cursor Failed
078N	Fetch from Cache Failed
078O	Add to Cache Failed
078P	Update to Cache Failed
078Q	Delete from Cache Failed
078R	Terminate of Cache Failed

Inserting Parameters for Error Messages in lpDS

Include the parameters cSuppressErrorMessage and szErrorMessageID in **lpDS** for error message processing. The functionality for each is as follows:

- cSuppressErrorMessage (SUPPS)

  Valid data is either 1 or 0. This parameter is required if **jdeErrorSet(...)** is used in the business function. When cSuppressErrorMessage is set to 1, do not set an error because **jdeErrorSet** will automatically display an error message.

- szErrorMessageID (DTAI)

  This 4–character string contains the error message ID value that is passed back by the business function. If an error occurs in the business function, szErrorMessageID contains that error number ID.

Note. You must initialize szErrorMessageID to 4 spaces at the beginning of the function. Failure to initialize can cause memory errors.

Example: Parameters in lpDS for an Error Message

This example includes the **lpDS** parameters, cSuppressErrorMessage, and szErrorMessageID:

```
if ((!IsStringBlank(lpDS->szErrorMessageID)) &&
     (lpDS->cSuppressErrorMessage != _J('1')))
 {
```



```
      jdeStrcpy ((JCHAR*) (lpDS->szErrorMessageID),
                 (const JCHAR*) (_J("0653")));
      jdeErrorSet (lpBhvrCom, lpVoid, (ID) IDERRcMethodofComputation_1,
                   lpDS->szErrorMessageID, (LPVOID) NULL);
      idReturnValue = ER_ERROR;
   }

/*************************************************************
 * Function Clean Up
 *************************************************************/
   return idReturnValue;
```

Initializing Behavior Errors

Business functions that use the JD Edwards EnterpriseOne database API are required to call the Initialize Behavior function before calling any of the database functions. Set error 078S if the Initialize Behavior function does not complete successfully.

Example: Initialize Behavior Error

This example illustrates an initialize behavior error:

```
/*************************************************************************
 * Initialize Behavior
 *************************************************************************/
idJDBReturn = JDB_InitBhvr(lpBhvrCom,
                           &hUser,
                           (JCHAR *) NULL,
                           JDEDB_COMMIT_AUTO);
if (idJDBReturn != JDEDB_PASSED)
{
   jdeStrcpy (lpDS->szErrorMessageID, _J("078S"));
   if (lpDS->cSuppressErrorMessage != _J('1'))
   {
      jdeErrorSet(lpBhvrCom, lpVoid, (ID)0, _J(078S), (LPVOID) NULL);
   }
   return ER_ERROR;
}
```

Using Text Substitution to Display Specific Error Messages

You can use the JD Edwards EnterpriseOne text substitution APIs for returning error messages within a business function. Text substitution is a flexible method for displaying a specific error message.

Text substitution is accomplished through the data dictionary. To use text substitution, you first must set up a data dictionary item that defines text substitution for the specific error message. A selection of error messages for JDB and JDE Cache have already been created and are listed in this chapter.

Error messages for cache and tables are critical in a configurable network computing (CNC) architecture. C programmers must set the appropriate error message when working with tables or cache APIs.

JDB API errors should substitute the name of the file against which the API failed. JDE cache API errors should substitute the name of the cache for which the API failed.

When calling errors that use text substitution, you must:

- Load a data structure with the information you want to substitute in the error message.
- Call **jdeErrorSet** to set the error.

Example: Text Substitution in an Error Message

This example uses text substitution in **JDB_OpenTable**:

```
/************************************************************
 * Open the General Ledger Table F0911
 ************************************************************/
eJDBReturn = JDB_OpenTable( hUser,
                            ID_F0911,
                            ID_F0911_DOC_TYPE__NUMBER___B,
                            idColF0911,
                            nNumColsF0911,
                            (JCHAR *)NULL,
                            &hRequestF0911);

if (eJDBReturn != JDEDB_PASSED)
{
   memset((void *)(&dsDE0022), 0x00, sizeof(dsDE0022));
   jdeStrncpy((JCHAR *)dsDE0022.szDescription,
              (const JCHAR *)(_J("F0911")),
              DIM(dsDE0022.szDescription)-1);
   jdeErrorSet (lpBhvrCom, lpVoid,(ID)0, _J("078D"), &dsDE0022);
}
```

Mapping Data Structure Errors with jdeCallObject

Any Business Function calling an external Business Function must use **jdeCallObject**. When using **jdeCallObject**, be sure to match the Error IDs correctly.

You need to match the Ids from the original Business Function with the Error Ids of the Business Function in **jdeCallObject**. A data structure is used in the **jdeCallObject** to accomplish this task.

```
/************************************************************
 * Variable declarations
 ************************************************************/
CALLMAP   cm_D0000026[2]  = {{IDERRmnDisplayExchgRate_62,
                            IDERRmnExchangeRate_2}};
ID        idReturnCode    = ER_SUCCESS;  /* Return Code */
/************************************************************
```



```
* Business Function structures
*******************************************************/
DSD0000026  dsD0000026    = {0};    /* Edit Tolerance */

 idReturnCode = jdeCallObject(_J("EditExchanbeRateTolerance"),
                              NULL,
                              lpBhvrCom,
                              lpVoid,
                              (void *)&dsD0000026,
                              (CALLMAP *)&cm_D0000026,
                              ND0000026,
                              (JCHAR *)NULL,
                              (JCHAR *)NULL,
                              (int)0);
```


CHAPTER 9

Understanding Data Dictionary Triggers

This chapter provides an overview of data dictionary triggers.

Data Dictionary Triggers

Data dictionary triggers are used to attach edit-and-display logic to data dictionary items. The application runtime engine executes the trigger associated with a data dictionary item at the time that the item is accessed in a form.

Custom data dictionary triggers are written in C or Named Event Rule (NER), and require a specific data structure in order to execute correctly. The custom trigger data structure is composed of three predefined members and one variable member. The predefined members are the same for every custom trigger. The variable member is different for each trigger, and it is created using the specific data element associated with the data dictionary item.

This table shows the order of the members in the data structure along with the alias and a description of each member.

Structure Member Name	Alias	Description
idBhvrErrorId	BHVRERRID	Used by the trigger function to return the error status (ER_ERROR or ER_SUCCESS) to the application.
szBehaviorEditString	BHVREDTST	Used by the application runtime engine to pass the value for the data dictionary field to the trigger function.
szDescription001	DL01	Used by the trigger function to return the description for the value to the application.
szHomeCompany, mnAddressNumber	HMCO, AN8	Used by the trigger function to set errors (CALLMAP field).

CHAPTER 10

Understanding Unicode Compliance Standards

This chapter discusses:

- Unicode compliance standards
- Unicode string functions
- Unicode memory functions
- Pointer arithmetic
- offsets
- **MATH_NUMERIC** APIs
- Third-party APIs
- Flat-file APIs

Unicode Compliance Standards

The Unicode Standard is the universal character-encoding scheme for written characters and text. It defines a consistent way of way of encoding multilingual text that enables the exchange of text data internationally and creates the foundation for global software.

Facts about Unicode:

- Unicode is a very large character set containing the characters of virtually every written language.
- Unicode uses two bytes per character.

  Up to 64,000 characters can be supported using two bytes. Unicode also has a mechanism called "surrogates," which uses pairs of two bytes to describe an additional one million characters.
- 0x00 is a valid byte in a character.

  For example, the character "A" is described as 0x00 0x41, which means that normal string functions, such as **strlen()** and **strcpy**, do not work with Unicode data.

Do not use the data type **char**. Instead, use **JCHAR** for Unicode characters and **ZCHAR** for non-Unicode characters. Use **ZCHAR** instead of **char** in a code that needs to interface with non-Unicode APIs.

Old Syntax No Longer Available	New Syntax Non-Unicode	New Syntax Unicode
Char	ZCHAR	JCHAR
char *, PSTR	ZCHAR*, PZSTR	JCHAR*, PJSTR

Old Syntax No Longer Available	New Syntax Non-Unicode	New Syntax Unicode
'A'	_Z('A')	_J('A')
"string"	_Z("string")	_J("string")

Unicode String Functions

Two versions of all string functions exist: one for Unicode and one for non-Unicode. Naming standards for Unicode and non-Unicode string functions are:

- **jdeSxxxxxx()** indicates a Unicode string function
- **jdeZSxxxx()** indicates a non-Unicode string function

Some of the replacement functions include:

Old String Functions	New String Functions Non-Unicode	New String Functions Unicode
strcpy()	jdeZStrcpy()	jdeStrcpy()
strlen()	jdeZStrlen()	jdeStrlen()
strstr()	jdeZStrstr()	jdeStrstr()
sprintf()	jdeZSprintf()	jdeSprintf()
strncpy()	jdeZStrncpy()	jdeStrncpy()

Note. The function **jdestrcpy()** was in use before the migration to Unicode. The Unicode slimer changed existing **jdestrcpy()** to **jdeStrncpyTerminate()**. Going forward, developers need to use **jdeStrncpyTerminate()** where they previously used **jdestrcpy()**.

Do not use traditional string functions, such as **strcpy**, **strlen**, and **printf**. All the **jdeStrxxxxxx** functions explicitly handle strings, so use character length instead of the **sizeof()** operator, which returns a byte count.

When using **jdeStrncpy()**, the third parameter is the number of characters, not the number of bytes.

The **DIM()** macro gives the number of characters of an array. Given "`JCHAR a[10];`", `DIM(a)` returns 10, while `sizeof(a)` returns 20. "`strncpy (a, b, sizeof (a));`" needs to become "`jdeStrncpy (a, b, DIM (a));`".

Example: Using Unicode String Functions

This example shows how to use Unicode string functions:

```
/*********************************************************************
 In this example jdeStrncpy replaces strncpy. Also sizeof is
 replaced by DIM.
 *********************************************************************/
```



```
/* Set key to F38112 */

/*Unicode Compliant*/
jdeStrncpy(dsKey1F38112.dxdcto,
          (const JCHAR *)(dsF4311ZDetail->pwdcto),
          DIM(dsKey1F38112.dxdcto) - 1);
```

Unicode Memory Functions

The **memset()** function changes memory byte by byte. For example, `memset (buf, ' ', sizeof (buf));` sets the 10 bytes pointed to by the first argument, buf, to the value 0x20, the space character. Since a Unicode character is 2 bytes, each character is set to 0x2020, which is the dagger character (†) in Unicode.

A new function, **jdeMemset()** sets memory character by character rather than byte by byte. This function takes a void pointer, a **JCHAR**, and the number of bytes to set. Use `jdeMemset (buf, _J (' '), sizeof (buf));` to set the Unicode string **buf** so that each character is 0x0020. When using **jdeMemset()**, the third parameter, **sizeof(buf)**, is the number of bytes, not characters.

Note. You can use **memset** when filling a memory block with NULL. For all other characters, use **jdeMemset**. You also can use **jdeMemset** for a NULL character.

Example: Using jdeMemset when Setting Characters to Values other than NULL

This example shows how to use **jdeMemset** when setting characters to values other than NULL:

```
/*********************************************************************
 In this example memset is replaced by jdeMemset. We need to change
 memset to jdeMemset because we are setting each character of the
 string to a value other than NULL. Also, because jdeMemset works in
 bytes, we cannot just subtract 1 from sizeof(szSubsidiaryBlank) to
 prevent the last character from being set to ' '. We must multiply
 1 by sizeof(JCHAR).
 *********************************************************************/

/*Unicode Compliant*/
jdeMemset((void *)(szSubsidiaryBlank), _J(' '),
          (sizeof(szSubsidiaryBlank) - (1*sizeof(JCHAR))));
```

Pointer Arithmetic

When advancing a **JCHAR** pointer, it is important to advance the pointer by the correct number. In the example, the intent is to initialize each member of an array consisting of **JCHAR** strings to blank. Inside the "For" loop, the pointer is advanced to point to the next member of the array of **JCHAR** strings after assigning a value to one of the members of the array. This is achieved by adding the maximum length of the string to the pointer. Since **pStringPtr** has been defined as a pointer to a **JCHAR**, adding **MAXSTRLENGTH** to **pStringPtr** results in **pStringPtr** pointing to the next member of the array of strings.

```
#define MAXSTRLENGTH 10
JCHAR             *pStringPtr;
LPMATH_NUMERIC    pmnPointerToF3007;
for(i=(iDayOfTheWeek+iNumberOfDaysInMonth);i<CALENDARDAYS;i++)
{
  FormatMathNumeric(pStringPtr, &pmnPointerToF3007[i]);
  pStringPtr = pStringPtr + MAXSTRLENGTH;
}
```

These tables illustrate the effect of adding **MAXSTRLENGTH** to **pStringPtr**. The top row in both tables contains memory locations; the bottom rows contain the contents of those memory locations.

The arrow indicates the memory location that **pStringPtr** points to before **MAXSTRLENGTH** is added to **pStringPtr**.

∇																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
00	49	00	52	00	20	00	20	00	20	00	20	00	20	00	20	00	20	00	20

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
00	49	00	52	00	20	00	20	00	20	00	20	00	20	00	20	00	20	00	20

The arrow indicates the memory location that **pStringPtr** points to after **MAXSTRLENGTH** is added to **pStringPtr**. Adding 10 to **pStringPtr** makes it move 20 bytes, as it has been declared of type **JCHAR**.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
00	52	00	53	00	54	00	20	00	20	00	20	00	20	00	20	00	20	00	20

∇																			
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
00	49	00	52	00	20	00	20	00	20	00	20	00	20	00	20	00	20	00	20

If **pStringPtr** is advanced by the value `MAXSTRLENGTH * sizeof(JCHAR)`, then **pStringPtr** advances twice as much as intended and results in memory corruption.

Offsets

When adding an offset to a pointer to derive the location of another variable or entity, it is important to determine the method in which the offset was initially created.

In this example, `lpKeyStruct->CacheKey[n].nOffset` is added to **lpData** to arrive at the location of a Cache Key segment. This offset was for the segment created using the ANSI C function **offsetof**, which returns the number of bytes. Therefore, to arrive at the location of Cache Key segment, cast the data structure pointer to type BYTE.

```
lpTemp1 = (BYTE *)lpData + lpKeyStruct->CacheKey[n].nOffset;
lpTemp2 = (BYTE *)lpKey  + lpKeyStruct->CacheKey[n].nOffset;
```

In a non-Unicode environment, **lpData** could have been cast to be of type **CHAR *** as character size is one Byte in a non-Unicode environment. In a Unicode environment, however, **lpData** has to be explicitly cast to be of type **(JCHAR *)** since size of a **JCHAR** is 2 bytes.

MATH_NUMERIC APIs

The string members of the **MATH_NUMERIC** data structure are in **ZCHAR** (non-Unicode) format. The JD Edwards EnterpriseOne Common Library API includes several functions that retrieve and manipulate these strings in both **JCHAR** (Unicode) and **ZCHAR** (non-Unicode) formats.

To retrieve the string value of a **MATH_NUMERIC** data type in **JCHAR** format, use the **FormatMathNumeric** API function. This example illustrates the use of this function:

```
/* Declare variables */
JCHAR      szJobNumber[MAXLEN_MATH_NUMERIC+1] = _J("\0");
/* Retrieve the string value of the job number */
FormatMathNumeric(szJobNumber, &lpDS->mnJobnumber);
```

To retrieve the string value of a MATH_NUMERIC data type in **ZCHAR** format, use the **jdeMathGetRawString** API function. This example illustrates the use of this function:

```
/* Declare variables */
ZCHAR      zzJobNumber[MAXLEN_MATH_NUMERIC+1] = _Z("\0");
/* Retrieve the string value of the job number */
zzJobNumber = jdeMathGetRawString(&lpDS->mnJobnumber);
```

Another commonly used **MATH_NUMERIC** API function is **jdeMathSetCurrencyCode**. This function is used to update the currency code member of a **MATH_NUMERIC** data structure. Two versions of this function exist: **jdeMathCurrencyCode** and **jdeMathCurrencyCodeUNI**. The **jdeMathCurrencyCode** function is used to update the currency code with a **ZCHAR** value, and **jdeMathCurrencyCodeUNI** is used to update the currency code with a **JCHAR** value. This example illustrates the use of these two functions:

```
/* Declare variables */
ZCHAR      zzCurrencyCode[4] = _Z("USD");
JCHAR      szCurrencyCode[4] = _J("USD");
/* Set the currency code using a ZCHAR value */
jdeMathSetCurrencyCode(&lpDs->mnAmount, (ZCHAR *) zzCurrencyCode);
/* Set the currency code using a JCHAR value /*
jdeMathSetCurrencyCodeUNI(&lpDS->mnAmount, (JCHAR *) szCurrencyCode);
```

Third-Party APIs

Some third-party program interfaces (APIs) do not support Unicode character strings. In these cases, you must convert character strings to non-Unicode format before calling the API, and convert them back to Unicode format for storage in JD Edwards EnterpriseOne. Use these guidelines when programming for a non-Unicode API:

- Declare a Unicode and a non-Unicode variable for each API string parameter.
- Convert the Unicode strings to non-Unicode strings before calling the API.
- Call the API passing the non-Unicode strings in the parameter list.
- Convert the returned non-Unicode strings to Unicode strings for storage in JD Edwards EnterpriseOne.

Example: Third-Party API

This example calls a third-party API named **GetStateName** that accepts a two-character state code and returns a 30-character state name:

```
/* Declare variables */
JCHAR  szStateCode[3] = _J("CO");  /* Unicode state code */
JCHAR  szStateName[31] = _J("\0");  /* Unicode state name */
ZCHAR  zzStateCode[3] = _Z("\0");  /* Non-Unicode state code */
ZCHAR  zzStateName[31] = _Z("\0");  /* Non-Unicode state name */
BOOL  bReturnStatus = FALSE;    /* API return flag */
/* Convert unicode strings to non-unicode strings */
jdeFromUnicode(zzStateCode, szStateCode, DIM(zzStateCode), NULL);
/* Call API */
bReturnStatus = GetStateName(zzStateCode, zzStateName);
/* Convert non-unicode strings to unicode strings for storage in
* JD Edwards EnterpriseOne */
jdeToUnicode(szStateName, zzStateName, DIM(szStateName), NULL);
```

Flat-File APIs

JD Edwards EnterpriseOne APIs such as **jdeFprintf()** convert data. This means that the default flat file I/O for character data is in Unicode. If the users of JD Edwards EnterpriseOne-generated flat files are not Unicode enabled, they will not be able to read the flat file correctly. Therefore, use an additional set of APIs.

An interactive application allows users to configure flat file encoding based on attributes such as application name, application version name, user name, and environment name. The API set includes these file I/O functions: **fwrite**/**fread**, **fprintf**/**fscanf**, **fputs**/**fgets**, and **fputc**/**fgetc**. The API converts the data using the code page specified in the configuration application. One additional parameter, **lpBhvrCom**, must be passed to the functions so that the conversion function can find the configuration for that application or version.

These new APIs only need to be called if a process outside of JD Edwards EnterpriseOne is writing or reading the flat file data. If the file is simply a work file or a debugging file and will be written and read by JD Edwards EnterpriseOne, use the non-converting APIs (for example, **jdeFprintf()**).

Example: Flat-File APIs

This example writes text to a flat file that would only be read by JD Edwards EnterpriseOne. Encoding in the file will be Unicode.

```
FILE *fp;
fp = jdeFopen(_J( c:/testBSFNZ.txt), _J(w+));
jdeFprintf(fp, _J("%s%d\n"), _J("Line "), 1);
jdeFclose(fp);
```

This example writes text to a flat file that would be read by third-party systems. Encoding in the file will be based on the encoding configured.

```
FILE *fp;
fp = jdeFopen(_J( c:/testBSFNZ.txt), _J(w+));
jdeFprintfConvert(lpBhvrCom, fp, _J("%s%d\n"), _J("Line "), 1);
jdeFclose(fp);
```

See Also

JD Edwards EnterpriseOne Tools 8.96 Interoperability Guide, “Using Flat Files”

CHAPTER 11

Understanding Standard Header and Source Files

This chapter provides an overview of standard header and standard source files.

Standard Header

Header files help the compiler properly create a business function. The C language contains 33 keywords. Everything else, such as **printf** and **getchar**, is a function. Functions are defined in header files that you include at the beginning of a business function. Without header files, the compiler does not recognize the functions and might return error messages.

This example shows the standard header for a business function source file:

```
/*****************************************************************
 *  Header File: BXXXXXXX.h
 *  Description: Generic Business Function Header File
 *    History:
 *     Date    Programmer SAR# - Description
 *     ---------- ---------- ---------------------------
 *  Author 03/15/2006           - Created
 *
 * Copyright (c) Oracle, 2006
 *
 * This unpublished material is proprietary to Oracle.
 * All rights reserved. The methods and
 * techniques described herein are considered trade secrets
 * and/or confidential. Reproduction or distribution, in whole
 * or in part, is forbidden except by express written permission
 * of Oracle.
 *****************************************************************/
#ifndef __BXXXXXXX_H
#define __BXXXXXXX_H
/*****************************************************************
 * Table Header Inclusions
 *****************************************************************/

/*****************************************************************
 * External Business Function Header Inclusions
 *****************************************************************/

/*****************************************************************
```

```
 * Global Definitions
 ****************************************************************/

/****************************************************************
 * Structure Definitions
 ****************************************************************/

/****************************************************************
 * DS Template Type Definitions
 ****************************************************************/

/****************************************************************
 * Source Preprocessor Definitions
 ****************************************************************/
#if defined (JDEBFRTN)
   #undef JDEBFRTN
#endif

#if defined (WIN32)
   #if defined (WIN32)
      #define JDEBFRTN(r) __declspec(dllexport) r
   #else
      #define JDEBFRTN(r) __declspec(dllimport) r
   #endif
#else
   #define JDEBFRTN(r) r
#endif
/****************************************************************
 * Business Function Prototypes
 ****************************************************************/
JDEBFRTN (ID) JDEBFWINAPI GenericBusinessFunction
                (LPBHVRCOM     lpBhvrCom,
                 LPVOID        lpVoid,
                 LPDSDXXXXXXXX lpDS);

/****************************************************************
 * Internal Function Prototypes
 ****************************************************************/
#endif  /* ___BXXXXXXX_H */
```

Business Function Name and Description

Use the Business Function Name and Description section to define the name of the business function, describe the business function, and maintain the modification log.

Copyright Notice

The Copyright section contains the Oracle copyright notice and must be included in each source file. Do not change this section.

Header Definition for a Business Function

The Header Definition section for a Business Function contains the "#define" of the business function. It is generated by the tool. Do not change this section.

Table Header Inclusions

The Table Header Inclusions section contains the include statements for the table headers associated with tables directly accessed by the business function.

External Business Function Header Inclusions

The External Business Function Header Inclusions section contains the include statements for the business function headers associated with externally defined business functions that are directly accessed by the business function.

Global Definitions

Use the Global Definitions section to define global constants used by the business function. Enter names in uppercase, separated by an underscore.

Structure Definitions

Define structures used by the business function in the Structure Definitions section. Structure names should be prefixed by the Source File Name to prevent conflicts with structures of the same name in other business functions.

See Chapter 2, “Understanding Naming Conventions,” page 3.

DS Template Type Definitions

The DS Template Type Definitions section defines the business functions contained in the source that correspond to the header. You generate the structure from the business function or data structure design window in Object Management Workbench. After you generate the structure, copy and paste it into this section.

Source Preprocessing Definitions

The Source Preprocessing Definitions section defines the entry point of the business function and includes the opening bracket required by C functions. Do not change this section.

Business Function Prototypes

Use the Business Function Prototypes section to prototype the functions defined in the source file.

Internal Function Prototypes

The Internal Function Prototypes section contains a description and parameters of the function.

See Chapter 2, “Understanding Naming Conventions,” page 3.

Standard Source

The source file contains instructions for the business function. These sections describe the sections of a standard source file.

A template generated for a standard source file when you create a JD Edwards EnterpriseOne business function appears in the following pages:

```
#include <jde.h>
#define bxxxxxxx_c
/*************************************************************
 *  Source File: bxxxxxxx
 *
 *  Description: Generic Business Function Source File
 *
 *    History:
 *     Date    Programmer SAR# - Description
 *    ----------------------------------------------------
 *  Author 06/06/2005           - Created
 *
 * Copyright (c) Oracle, 2005
 *
 * This unpublished material is proprietary to Oracle.
 * All rights reserved. The methods and techniques described
 * herein are considered trade secrets and/or confidential.
 * Reproduction or distribution, in whole or in part, is
 * forbidden except by express written permission of
 * Oracle.
 *************************************************************/
/*************************************************************
 * Notes:
 *
 *************************************************************/

#include <bxxxxxxx.h>
/*************************************************************
 * Global Definitions

*************************************************************/

/*************************************************************
 * Business Function: GenericBusinessFunction
 *
 *   Description: Generic Business Function
 *
 *   Parameters:
 *   LPBHVRCOM  lpBhvrCom Business Function Communications
 *   LPVOID    lpVoid  Void Parameter - DO NOT USE!
 *   LPDSDXXXXXXX lpDS   Parameter Data Structure Pointer
 *
 *************************************************************/
JDEBFRTN (ID) JDEBFWINAPI GenericBusinessFunction
                 (LPBHVRCOM     lpBhvrCom,
                  LPVOID        lpVoid,
                  LPDSDXXXXXXXX lpDS)
```



```
{
 /*********************************************************
  * Variable declarations
  *********************************************************/

 /*********************************************************
  * Declare structures
  *********************************************************/

 /*********************************************************
  * Declare pointers
  *********************************************************/

 /*********************************************************
  * Check for NULL pointers
  *********************************************************/
 if ((lpBhvrCom == (LPBHVRCOM) NULL) ||
     (lPVoid  == (LPVOID)  NULL) ||
     (lpDS   == (LPDSDXXXXXXXX) NULL))
 {
    jdeErrorSet (lpBhvrCom, lpVoid, (ID) 0,
                 4363, (LPVOID) NULL);
    return ER_ERROR;
 }
 /*********************************************************
  * Set pointers
  *********************************************************/

/*********************************************************
  * Main Processing
  *********************************************************/

 /*********************************************************
  * Function Clean Up
  *********************************************************/

 return (ER_SUCCESS);
}
/* Internal function comment block */
/*********************************************************
 *  Function: Ixxxxxxx_a // Replace xxxxxxx with source file
 *             // number
 *             // and a with the function name
 *   Notes:
 *
 *  Returns:
 *
 * Parameters:
*********************************************************/
```

Business Function Name and Description

Use this section to maintain the name and description of the business function. Also use this section to maintain the modification log.

Copyright Notice

The Copyright section contains the Oracle copyright notice and must be included in each source file. Do not make any changes to this section.

Notes

Use the Notes section to include information for anyone who might review the code in the future. For example, describe any peculiarities associated with the business function or any special logic.

Global Definitions

Use the Global Definitions section to define global constants used by the business function.

Header File for Associated Business Function

In the Header File for Associated Business Function section, include the header file associated with the business function using #include. If you need to include additional header files in the source, place them here.

Business Function Header

The Business Function Header section contains a description of each of the parameters used by the business function. Do not make any changes to this section.

Variable Declarations

The Variable Declarations section defines all required function variables. For ease of use, define the variables sequentially by type.

See Chapter 2, "Understanding Naming Conventions," page 3.

Declare Structures

Define any structures that are required by the function in the Declare Structures section.

Pointers

If any pointers are required by the function, define them in the Pointers section. Name the pointer so that it reflects the structure to which it is pointing. For example, lpDS1100 is pointing to the structure DS1100.

Check for NULL Pointers

The Check for NULL Pointers section checks for parameter pointers that are NULL. Do not change this section.

Set Pointers

Use the Set Pointers section if you did not initialize the variables when declaring them. You must assign values to all pointers that you define.

Main Processing

Use the Main Processing section to write the code.

Function Clean Up

Use the Function Clean Up section to release any allocated memory.

Internal Function Comment Block

The Internal Function Comment Block section contains a description and parameters of the function.

See Chapter 2, “Understanding Naming Conventions,” page 3.

Glossary of JD Edwards EnterpriseOne Terms

activity	A scheduling entity in JD Edwards EnterpriseOne tools that represents a designated amount of time on a calendar.
activity rule	The criteria by which an object progresses from one given point to the next in a flow.
add mode	A condition of a form that enables users to input data.
Advanced Planning Agent (APAg)	A JD Edwards EnterpriseOne tool that can be used to extract, transform, and load enterprise data. APAg supports access to data sources in the form of rational databases, flat file format, and other data or message encoding, such as XML.
application server	A server in a local area network that contains applications shared by network clients.
as if processing	A process that enables you to view currency amounts as if they were entered in a currency different from the domestic and foreign currency of the transaction.
alternate currency	A currency that is different from the domestic currency (when dealing with a domestic-only transaction) or the domestic and foreign currency of a transaction. In JD Edwards EnterpriseOne Financial Management, alternate currency processing enables you to enter receipts and payments in a currency other than the one in which they were issued.
as of processing	A process that is run as of a specific point in time to summarize transactions up to that date. For example, you can run various JD Edwards EnterpriseOne reports as of a specific date to determine balances and amounts of accounts, units, and so on as of that date.
back-to-back process	A process in JD Edwards EnterpriseOne Supply Management that contains the same keys that are used in another process.
batch processing	A process of transferring records from a third-party system to JD Edwards EnterpriseOne. In JD Edwards EnterpriseOne Financial Management, batch processing enables you to transfer invoices and vouchers that are entered in a system other than JD Edwards EnterpriseOne to JD Edwards EnterpriseOne Accounts Receivable and JD Edwards EnterpriseOne Accounts Payable, respectively. In addition, you can transfer address book information, including customer and supplier records, to JD Edwards EnterpriseOne.
batch server	A server that is designated for running batch processing requests. A batch server typically does not contain a database nor does it run interactive applications.
batch-of-one immediate	A transaction method that enables a client application to perform work on a client workstation, then submit the work all at once to a server application for further processing. As a batch process is running on the server, the client application can continue performing other tasks. See also direct connect and store-and-forward.
business function	A named set of user-created, reusable business rules and logs that can be called through event rules. Business functions can run a transaction or a subset of a transaction (check inventory, issue work orders, and so on). Business functions also contain the application programming interfaces (APIs) that enable them to be called from a form, a database trigger, or a non-JD Edwards EnterpriseOne application. Business functions can be combined with other business functions, forms, event rules,

	and other components to make up an application. Business functions can be created through event rules or third-generation languages, such as C. Examples of business functions include Credit Check and Item Availability.
business function event rule	See named event rule (NER).
business view	A means for selecting specific columns from one or more JD Edwards EnterpriseOne application tables whose data is used in an application or report. A business view does not select specific rows, nor does it contain any actual data. It is strictly a view through which you can manipulate data.
central objects merge	A process that blends a customer's modifications to the objects in a current release with objects in a new release.
central server	A server that has been designated to contain the originally installed version of the software (central objects) for deployment to client computers. In a typical JD Edwards EnterpriseOne installation, the software is loaded on to one machine—the central server. Then, copies of the software are pushed out or downloaded to various workstations attached to it. That way, if the software is altered or corrupted through its use on workstations, an original set of objects (central objects) is always available on the central server.
charts	Tables of information in JD Edwards EnterpriseOne that appear on forms in the software.
connector	Component-based interoperability model that enables third-party applications and JD Edwards EnterpriseOne to share logic and data. The JD Edwards EnterpriseOne connector architecture includes Java and COM connectors.
contra/clearing account	A general ledger account in JD Edwards EnterpriseOne Financial Management that is used by the system to offset (balance) journal entries. For example, you can use a contra/clearing account to balance the entries created by allocations in JD Edwards EnterpriseOne Financial Management.
Control Table Workbench	An application that, during the Installation Workbench processing, runs the batch applications for the planned merges that update the data dictionary, user-defined codes, menus, and user override tables.
control tables merge	A process that blends a customer's modifications to the control tables with the data that accompanies a new release.
cost assignment	The process in JD Edwards EnterpriseOne Advanced Cost Accounting of tracing or allocating resources to activities or cost objects.
cost component	In JD Edwards EnterpriseOne Manufacturing, an element of an item's cost (for example, material, labor, or overhead).
cross segment edit	A logic statement that establishes the relationship between configured item segments. Cross segment edits are used to prevent ordering of configurations that cannot be produced.
currency restatement	The process of converting amounts from one currency into another currency, generally for reporting purposes. You can use the currency restatement process, for example, when many currencies must be restated into a single currency for consolidated reporting.
database server	A server in a local area network that maintains a database and performs searches for client computers.
Data Source Workbench	An application that, during the Installation Workbench process, copies all data sources that are defined in the installation plan from the Data Source Master and Table and Data Source Sizing tables in the Planner data source to the system-release number data source. It also updates the Data Source Plan detail record to reflect completion.

date pattern A calendar that represents the beginning date for the fiscal year and the ending date for each period in that year in standard and 52-period accounting.

denominated-in currency The company currency in which financial reports are based.

deployment server A server that is used to install, maintain, and distribute software to one or more enterprise servers and client workstations.

detail information Information that relates to individual lines in JD Edwards EnterpriseOne transactions (for example, voucher pay items and sales order detail lines).

direct connect A transaction method in which a client application communicates interactively and directly with a server application.

See also batch-of-one immediate and store-and-forward.

Do Not Translate (DNT) A type of data source that must exist on the iSeries because of BLOB restrictions.

dual pricing The process of providing prices for goods and services in two currencies.

edit code A code that indicates how a specific value for a report or a form should appear or be formatted. The default edit codes that pertain to reporting require particular attention because they account for a substantial amount of information.

edit mode A condition of a form that enables users to change data.

edit rule A method used for formatting and validating user entries against a predefined rule or set of rules.

Electronic Data Interchange (EDI) An interoperability model that enables paperless computer-to-computer exchange of business transactions between JD Edwards EnterpriseOne and third-party systems. Companies that use EDI must have translator software to convert data from the EDI standard format to the formats of their computer systems.

embedded event rule An event rule that is specific to a particular table or application. Examples include form-to-form calls, hiding a field based on a processing option value, and calling a business function. Contrast with the business function event rule.

Employee Work Center A central location for sending and receiving all JD Edwards EnterpriseOne messages (system and user generated), regardless of the originating application or user. Each user has a mailbox that contains workflow and other messages, including Active Messages.

enterprise server A server that contains the database and the logic for JD Edwards EnterpriseOne.

EnterpriseOne object A reusable piece of code that is used to build applications. Object types include tables, forms, business functions, data dictionary items, batch processes, business views, event rules, versions, data structures, and media objects.

EnterpriseOne process A software process that enables JD Edwards EnterpriseOne clients and servers to handle processing requests and run transactions. A client runs one process, and servers can have multiple instances of a process. JD Edwards EnterpriseOne processes can also be dedicated to specific tasks (for example, workflow messages and data replication) to ensure that critical processes don't have to wait if the server is particularly busy.

Environment Workbench An application that, during the Installation Workbench process, copies the environment information and Object Configuration Manager tables for each environment from the Planner data source to the system-release number data source. It also updates the Environment Plan detail record to reflect completion.

escalation monitor A batch process that monitors pending requests or activities and restarts or forwards them to the next step or user after they have been inactive for a specified amount of time.

event rule A logic statement that instructs the system to perform one or more operations based on an activity that can occur in a specific application, such as entering a form or exiting a field.

facility An entity within a business for which you want to track costs. For example, a facility might be a warehouse location, job, project, work center, or branch/plant. A facility is sometimes referred to as a "business unit."

fast path A command prompt that enables the user to move quickly among menus and applications by using specific commands.

file server A server that stores files to be accessed by other computers on the network. Unlike a disk server, which appears to the user as a remote disk drive, a file server is a sophisticated device that not only stores files, but also manages them and maintains order as network users request files and make changes to these files.

final mode The report processing mode of a processing mode of a program that updates or creates data records.

FTP server A server that responds to requests for files via file transfer protocol.

header information Information at the beginning of a table or form. Header information is used to identify or provide control information for the group of records that follows.

interface table See Z table.

integration server A server that facilitates interaction between diverse operating systems and applications across internal and external networked computer systems.

integrity test A process used to supplement a company's internal balancing procedures by locating and reporting balancing problems and data inconsistencies.

interoperability model A method for third-party systems to connect to or access JD Edwards EnterpriseOne.

in-your-face-error In JD Edwards EnterpriseOne, a form-level property which, when enabled, causes the text of application errors to appear on the form.

IServer service This internet server service resides on the web server and is used to speed up delivery of the Java class files from the database to the client.

jargon An alternative data dictionary item description that JD Edwards EnterpriseOne appears based on the product code of the current object.

Java application server A component-based server that resides in the middle-tier of a server-centric architecture. This server provides middleware services for security and state maintenance, along with data access and persistence.

JDBNET A database driver that enables heterogeneous servers to access each other's data.

JDEBASE Database Middleware A JD Edwards EnterpriseOne proprietary database middleware package that provides platform-independent APIs, along with client-to-server access.

JDECallObject An API used by business functions to invoke other business functions.

jde.ini A JD Edwards EnterpriseOne file (or member for iSeries) that provides the runtime settings required for JD Edwards EnterpriseOne initialization. Specific versions of the file or member must reside on every machine running JD Edwards EnterpriseOne. This includes workstations and servers.

JDEIPC Communications programming tools used by server code to regulate access to the same data in multiprocess environments, communicate and coordinate between processes, and create new processes.

jde.log The main diagnostic log file of JD Edwards EnterpriseOne. This file is always located in the root directory on the primary drive and contains status and error messages from the startup and operation of JD Edwards EnterpriseOne.

JDENET A JD Edwards EnterpriseOne proprietary communications middleware package. This package is a peer-to-peer, message-based, socket-based, multiprocess communications middleware solution. It handles client-to-server and server-to-server communications for all JD Edwards EnterpriseOne supported platforms.

Location Workbench An application that, during the Installation Workbench process, copies all locations that are defined in the installation plan from the Location Master table in the Planner data source to the system data source.

logic server A server in a distributed network that provides the business logic for an application program. In a typical configuration, pristine objects are replicated on to the logic server from the central server. The logic server, in conjunction with workstations, actually performs the processing required when JD Edwards EnterpriseOne software runs.

MailMerge Workbench An application that merges Microsoft Word 6.0 (or higher) word-processing documents with JD Edwards EnterpriseOne records to automatically print business documents. You can use MailMerge Workbench to print documents, such as form letters about verification of employment.

master business function (MBF) An interactive master file that serves as a central location for adding, changing, and updating information in a database. Master business functions pass information between data entry forms and the appropriate tables. These master functions provide a common set of functions that contain all of the necessary default and editing rules for related programs. MBFs contain logic that ensures the integrity of adding, updating, and deleting information from databases.

master table See published table.

matching document A document associated with an original document to complete or change a transaction. For example, in JD Edwards EnterpriseOne Financial Management, a receipt is the matching document of an invoice, and a payment is the matching document of a voucher.

media storage object Files that use one of the following naming conventions that are not organized into table format: Gxxx, xxxGT, or GTxxx.

message center A central location for sending and receiving all JD Edwards EnterpriseOne messages (system and user generated), regardless of the originating application or user.

messaging adapter An interoperability model that enables third-party systems to connect to JD Edwards EnterpriseOne to exchange information through the use of messaging queues.

messaging server A server that handles messages that are sent for use by other programs using a messaging API. Messaging servers typically employ a middleware program to perform their functions.

named event rule (NER) Encapsulated, reusable business logic created using event rules, rather that C programming. NERs are also called business function event rules. NERs can be reused in multiple places by multiple programs. This modularity lends itself to streamlining, reusability of code, and less work.

nota fiscal In Brazil, a legal document that must accompany all commercial transactions for tax purposes and that must contain information required by tax regulations.

nota fiscal factura In Brazil, a nota fiscal with invoice information.

See also *nota fiscal.*

Object Configuration Manager (OCM) In JD Edwards EnterpriseOne, the object request broker and control center for the runtime environment. OCM keeps track of the runtime locations for business functions, data, and batch applications. When one of these objects is called, OCM directs access to it using defaults and overrides for a given environment and user.

Object Librarian A repository of all versions, applications, and business functions reusable in building applications. Object Librarian provides check-out and check-in capabilities for developers, and it controls the creation, modification, and use of JD Edwards EnterpriseOne objects. Object Librarian supports multiple environments (such as production and development) and enables objects to be easily moved from one environment to another.

Object Librarian merge A process that blends any modifications to the Object Librarian in a previous release into the Object Librarian in a new release.

Open Data Access (ODA) An interoperability model that enables you to use SQL statements to extract JD Edwards EnterpriseOne data for summarization and report generation.

Output Stream Access (OSA) An interoperability model that enables you to set up an interface for JD Edwards EnterpriseOne to pass data to another software package, such as Microsoft Excel, for processing.

package JD Edwards EnterpriseOne objects are installed to workstations in packages from the deployment server. A package can be compared to a bill of material or kit that indicates the necessary objects for that workstation and where on the deployment server the installation program can find them. It is point-in-time snapshot of the central objects on the deployment server.

package build A software application that facilitates the deployment of software changes and new applications to existing users. Additionally, in JD Edwards EnterpriseOne, a package build can be a compiled version of the software. When you upgrade your version of the ERP software, for example, you are said to take a package build.

Consider the following context: "Also, do not transfer business functions into the production path code until you are ready to deploy, because a global build of business functions done during a package build will automatically include the new functions." The process of creating a package build is often referred to, as it is in this example, simply as "a package build."

package location The directory structure location for the package and its set of replicated objects. This is usually \\deployment server\release\path_code\package\package name. The subdirectories under this path are where the replicated objects for the package are placed. This is also referred to as where the package is built or stored.

Package Workbench An application that, during the Installation Workbench process, transfers the package information tables from the Planner data source to the system-release number data source. It also updates the Package Plan detail record to reflect completion.

planning family A means of grouping end items whose similarity of design and manufacture facilitates being planned in aggregate.

preference profile The ability to define default values for specified fields for a user-defined hierarchy of items, item groups, customers, and customer groups.

print server The interface between a printer and a network that enables network clients to connect to the printer and send their print jobs to it. A print server can be a computer, separate hardware device, or even hardware that resides inside of the printer itself.

pristine environment A JD Edwards EnterpriseOne environment used to test unaltered objects with JD Edwards EnterpriseOne demonstration data or for training classes. You must have this environment so that you can compare pristine objects that you modify.

processing option A data structure that enables users to supply parameters that regulate the running of a batch program or report. For example, you can use processing options to specify default values for certain fields, to determine how information appears or is printed, to specify date ranges, to supply runtime values that regulate program execution, and so on.

production environment A JD Edwards EnterpriseOne environment in which users operate EnterpriseOne software.

production-grade file server A file server that has been quality assurance tested and commercialized and that is usually provided in conjunction with user support services.

program temporary fix (PTF) A representation of changes to JD Edwards EnterpriseOne software that your organization receives on magnetic tapes or disks.

project In JD Edwards EnterpriseOne, a virtual container for objects being developed in Object Management Workbench.

promotion path The designated path for advancing objects or projects in a workflow. The following is the normal promotion cycle (path):

11>21>26>28>38>01

In this path, *11* equals new project pending review, *21* equals programming, *26* equals QA test/review, *28* equals QA test/review complete, *38* equals in production, *01* equals complete. During the normal project promotion cycle, developers check objects out of and into the development path code and then promote them to the prototype path code. The objects are then moved to the productions path code before declaring them complete.

proxy server A server that acts as a barrier between a workstation and the internet so that the enterprise can ensure security, administrative control, and caching service.

published table Also called a master table, this is the central copy to be replicated to other machines. Residing on the publisher machine, the F98DRPUB table identifies all of the published tables and their associated publishers in the enterprise.

publisher The server that is responsible for the published table. The F98DRPUB table identifies all of the published tables and their associated publishers in the enterprise.

pull replication One of the JD Edwards EnterpriseOne methods for replicating data to individual workstations. Such machines are set up as pull subscribers using JD Edwards EnterpriseOne data replication tools. The only time that pull subscribers are notified of changes, updates, and deletions is when they request such information. The request is in the form of a message that is sent, usually at startup, from the pull subscriber to the server machine that stores the F98DRPCN table.

QBE An abbreviation for query by example. In JD Edwards EnterpriseOne, the QBE line is the top line on a detail area that is used for filtering data.

real-time event A service that uses system calls to capture JD Edwards EnterpriseOne transactions as they occur and to provide notification to third-party software, end users, and other JD Edwards EnterpriseOne systems that have requested notification when certain transactions occur.

refresh A function used to modify JD Edwards EnterpriseOne software, or subset of it, such as a table or business data, so that it functions at a new release or cumulative update level, such as B73.2 or B73.2.1.

replication server A server that is responsible for replicating central objects to client machines.

quote order In JD Edwards Procurement and Subcontract Management, a request from a supplier for item and price information from which you can create a purchase order.

	In JD Edwards Sales Order Management, item and price information for a customer who has not yet committed to a sales order.
selection	Found on JD Edwards EnterpriseOne menus, a selection represents functions that you can access from a menu. To make a selection, type the associated number in the Selection field and press Enter.
Server Workbench	An application that, during the Installation Workbench process, copies the server configuration files from the Planner data source to the system-release number data source. It also updates the Server Plan detail record to reflect completion.
spot rate	An exchange rate entered at the transaction level. This rate overrides the exchange rate that is set up between two currencies.
Specification merge	A merge that comprises three merges: Object Librarian merge, Versions List merge, and Central Objects merge. The merges blend customer modifications with data that accompanies a new release.
specification	A complete description of a JD Edwards EnterpriseOne object. Each object has its own specification, or name, which is used to build applications.
Specification Table Merge Workbench	An application that, during the Installation Workbench process, runs the batch applications that update the specification tables.
store-and-forward	The mode of processing that enables users who are disconnected from a server to enter transactions and then later connect to the server to upload those transactions.
subscriber table	Table F98DRSUB, which is stored on the publisher server with the F98DRPUB table and identifies all of the subscriber machines for each published table.
supplemental data	Any type of information that is not maintained in a master file. Supplemental data is usually additional information about employees, applicants, requisitions, and jobs (such as an employee's job skills, degrees, or foreign languages spoken). You can track virtually any type of information that your organization needs. For example, in addition to the data in the standard master tables (the Address Book Master, Customer Master, and Supplier Master tables), you can maintain other kinds of data in separate, generic databases. These generic databases enable a standard approach to entering and maintaining supplemental data across JD Edwards EnterpriseOne systems.
table access management (TAM)	The JD Edwards EnterpriseOne component that handles the storage and retrieval of use-defined data. TAM stores information, such as data dictionary definitions; application and report specifications; event rules; table definitions; business function input parameters and library information; and data structure definitions for running applications, reports, and business functions.
Table Conversion Workbench	An interoperability model that enables the exchange of information between JD Edwards EnterpriseOne and third-party systems using non-JD Edwards EnterpriseOne tables.
table conversion	An interoperability model that enables the exchange of information between JD Edwards EnterpriseOne and third-party systems using non-JD Edwards EnterpriseOne tables.
table event rules	Logic that is attached to database triggers that runs whenever the action specified by the trigger occurs against the table. Although JD Edwards EnterpriseOne enables event rules to be attached to application events, this functionality is application specific. Table event rules provide embedded logic at the table level.
terminal server	A server that enables terminals, microcomputers, and other devices to connect to a network or host computer or to devices attached to that particular computer.

three-tier processing The task of entering, reviewing and approving, and posting batches of transactions in JD Edwards EnterpriseOne.

three-way voucher match In JD Edwards Procurement and Subcontract Management, the process of comparing receipt information to supplier's invoices to create vouchers. In a three-way match, you use the receipt records to create vouchers.

transaction processing (TP) monitor A monitor that controls data transfer between local and remote terminals and the applications that originated them. TP monitors also protect data integrity in the distributed environment and may include programs that validate data and format terminal screens.

transaction set An electronic business transaction (electronic data interchange standard document) made up of segments.

trigger One of several events specific to data dictionary items. You can attach logic to a data dictionary item that the system processes automatically when the event occurs.

triggering event A specific workflow event that requires special action or has defined consequences or resulting actions.

two-way voucher match In JD Edwards Procurement and Subcontract Management, the process of comparing purchase order detail lines to the suppliers' invoices to create vouchers. You do not record receipt information.

User Overrides merge Adds new user override records into a customer's user override table.

variance In JD Edwards Capital Asset Management, the difference between revenue generated by a piece of equipment and costs incurred by the equipment.

In JD Edwards EnterpriseOne Project Costing and JD Edwards EnterpriseOne Manufacturing, the difference between two methods of costing the same item (for example, the difference between the frozen standard cost and the current cost is an engineering variance). Frozen standard costs come from the Cost Components table, and the current costs are calculated using the current bill of material, routing, and overhead rates.

Version List merge The Versions List merge preserves any non-XJDE and non-ZJDE version specifications for objects that are valid in the new release, as well as their processing options data.

visual assist Forms that can be invoked from a control via a trigger to assist the user in determining what data belongs in the control.

vocabulary override An alternate description for a data dictionary item that appears on a specific JD Edwards EnterpriseOne form or report.

wchar_t An internal type of a wide character. It is used for writing portable programs for international markets.

web application server A web server that enables web applications to exchange data with the back-end systems and databases used in eBusiness transactions.

web server A server that sends information as requested by a browser, using the TCP/IP set of protocols. A web server can do more than just coordination of requests from browsers; it can do anything a normal server can do, such as house applications or data. Any computer can be turned into a web server by installing server software and connecting the machine to the internet.

Windows terminal server A multiuser server that enables terminals and minimally configured computers to display Windows applications even if they are not capable of running Windows software themselves. All client processing is performed centrally at the Windows

terminal server and only display, keystroke, and mouse commands are transmitted over the network to the client terminal device.

workbench A program that enables users to access a group of related programs from a single entry point. Typically, the programs that you access from a workbench are used to complete a large business process. For example, you use the JD Edwards EnterpriseOne Payroll Cycle Workbench (P07210) to access all of the programs that the system uses to process payroll, print payments, create payroll reports, create journal entries, and update payroll history. Examples of JD Edwards EnterpriseOne workbenches include Service Management Workbench (P90CD020), Line Scheduling Workbench (P3153), Planning Workbench (P13700), Auditor's Workbench (P09E115), and Payroll Cycle Workbench.

work day calendar In JD Edwards EnterpriseOne Manufacturing, a calendar that is used in planning functions that consecutively lists only working days so that component and work order scheduling can be done based on the actual number of work days available. A work day calendar is sometimes referred to as planning calendar, manufacturing calendar, or shop floor calendar.

workflow The automation of a business process, in whole or in part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules.

workgroup server A server that usually contains subsets of data replicated from a master network server. A workgroup server does not perform application or batch processing.

XAPI events A service that uses system calls to capture JD Edwards EnterpriseOne transactions as they occur and then calls third-party software, end users, and other JD Edwards EnterpriseOne systems that have requested notification when the specified transactions occur to return a response.

XML CallObject An interoperability capability that enables you to call business functions.

XML Dispatch An interoperability capability that provides a single point of entry for all XML documents coming into JD Edwards EnterpriseOne for responses.

XML List An interoperability capability that enables you to request and receive JD Edwards EnterpriseOne database information in chunks.

XML Service An interoperability capability that enables you to request events from one JD Edwards EnterpriseOne system and receive a response from another JD Edwards EnterpriseOne system.

XML Transaction An interoperability capability that enables you to use a predefined transaction type to send information to or request information from JD Edwards EnterpriseOne. XML transaction uses interface table functionality.

XML Transaction Service (XTS) Transforms an XML document that is not in the JD Edwards EnterpriseOne format into an XML document that can be processed by JD Edwards EnterpriseOne. XTS then transforms the response back to the request originator XML format.

Z event A service that uses interface table functionality to capture JD Edwards EnterpriseOne transactions and provide notification to third-party software, end users, and other JD Edwards EnterpriseOne systems that have requested to be notified when certain transactions occur.

Z table A working table where non-JD Edwards EnterpriseOne information can be stored and then processed into JD Edwards EnterpriseOne. Z tables also can be used to retrieve JD Edwards EnterpriseOne data. Z tables are also known as interface tables.

Z transaction Third-party data that is properly formatted in interface tables for updating to the JD Edwards EnterpriseOne database.

Index