PeopleSoft

EnterpriseOne Development Standards Business Function Programming 8.9 PeopleBook

September 2003

EnterpriseOne
Development Standards Business Function Programming 8.9 PeopleBook
SKU REL9EBS0309

Table of Contents

Business Function Programming Standards

A business function is an integral part of the J.D. Edwards software tool set. A business function allows application developers to attach custom functionality to application and batch processing events.

A standard method for creating C code that performs a specific action does not exist. C code can be as unique as the individual who programmed and compiled the code. This guide provides standards for coding business functions that are efficient, readable, and easy to maintain.

Program Flow

The program flow of a C function should be from the top down, modularizing the code segments. For readability and maintenance purposes, divide code into logical units.

A business function is also referred to as a J.D. Edwards business function because it is partially created using the J.D. Edwards software tools. A business function is checked in and checked out so that it is available to other applications and business functions.

An internal function can only be used within the source file. No other source file should access an internal function. Internal functions are created for modularity of a common routine that is called by the business function.

Naming Conventions

Standardized naming conventions ensure a consistent approach to identifying function objects and function sections.

Source and Header File Names

Source and header file names can be a maximum of 8 characters and should be formatted as follows:

bxxyyyyy

Where:

b = Source or header file

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.

The following 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 follows:

Ixxxxxxx_a

Where:

I = An internal function

xxxxxxx = The source file name

a = The function description. Function descriptions can be up to 32 characters in length. 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 follows:

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 the following table:

Prefix	Description
c	JCHAR
sz	NULL-terminated JCHAR string
z	ZCHAR
zz	NULL-terminated ZCHAR string
n	short
l	long
b	Boolean
mn	MATHNUMERIC
jd	JDEDATE
lp	long pointer
i	integer
by	byte
ul	unsigned long (identifier)
us	unsigned Short
ds	data structures
h	handle
e	enumerated types
id	id long integer, JDE data type for returns

Example: Hungarian Notation for Variable Names

The following 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 follows:

DxxyyyyA

Where:

D = data structure

xx (second two digits) = The system code, such as:

01 - Address Book

04 - 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 follows:

D050575A

See Also

- *Creating a Business Function Data Structure* in the *Development Tools Guide*

Readability

Readable code is easier to debug and maintain. The following section presents guidelines for making your 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 the following 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 the following 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

The following 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 off. Then, each time you press the Tab key, three spaces are inserted rather than the tab character. Turn on auto-indentation.

Example: Indenting Code

The following is 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 the following 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 the following 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

The following 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

Th following 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 the following 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

The following example shows how to handle multiple logical expressions:

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

{
   statements
}
```

Declaring and Initializing Variables and Data Structures

Variables and data structures must be defined and initialized before they can be used to store data. This section 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

The following 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

The following 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 following example for using a typedef statement for a data structure.

Example: Using Typedef for a User-Defined Data Structure

The following 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 the following 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.

See Also

- *Business Function Prototypes* in the *Standard Header* section of the *Development Standards Business Function Programming Guide*
- *Standard Header* in the *Development Standards Business Function Programming Guide*

Example: Creating a Business Function Prototype

The following 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

The following 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

The following 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

The following 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.

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

Use the following guidelines when declaring and initializing variables:

- Declare variables using the following 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 format example above) 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 memset immediately after the declaration section.
- Some 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.

See Also

- *Initializing Variables* in the *Development Standards Business Function Programming Guide* for details about initializing variables

Example: Initializing Variables

The following 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 the following 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

The following 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 presents the requirements for standard variables, including flag variable, input and output parameters, and fetch variables.

Flag Variables

When creating flag variables, use the following 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.

Examples of flag variables are listed below, with a brief description of how each is used:

bIsMemoryAllocated	Apply to memory allocation
bIsLinkListEmpty	Link List

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:

cReturnPointer	When allocating memory and returning GENLNG
cErrorCode	Based on cCallType, cErrorCode returns a 1 when it fails or 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.

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.

idJDEDBResult	APIs or J.D. Edwards 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 for defining the Table ID.
idIndexXXXXID	Where XXXX is the table name, such as F4101 or F41021, the variable used for defining 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

The following 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;
 DSX51013_F0901 dsF0901;

/************************************************************
 *  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
   }
}
```

General Coding Guidelines

This section contains guidelines for writing good code. It addresses using function calls, allocating and releasing memory, using hRequest and hUser, typecasting, comparison testing, copying strings with jdeStrcpy versus jdeStrncpy, inserting function exit points, and terminating a function.

Using Function Calls

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

- Always put a space between each parameter.
- 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

The following 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

Never call an internal business function by another business function. When code is copied from one source to another, change all of the internal function names to match the current internal naming standard for the function. Internal functions from other business functions can be included from another source code, and must never be used outside the current source code in the Client/Server paradigm.

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 the address for a pointer 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 is maintained by J.D. Edwards software tools. The index to a position in the array is a long integer type or ID. Use the GENLNG data dictionary object in your business function data structure to pass this index in or out of the business function.

Store 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

The following example illustrates how to store an address in an array:

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

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



}
```

Retrieve 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

The following example retrieves an address from an array:

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

Remove 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

The following 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 Free Ptr To Data Structure, B4000640, to release memory through event rule logic.

Example: Allocating and Releasing Memory within a Business Function

The following 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 hUser and hRequest. 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

The following 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 versus 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 less one for 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

The following 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 allows 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

The following 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.

Portability

Portability is the ability to run a program on more than one computer without modifying it. J.D. Edwards 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 the following:

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

Ideally, industry standards allow 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 the following 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 it is not portable, 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 the offset to retrieve the object 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).

Common Server Build Errors and Warnings

ERP 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 the following examples.

Example: C Comments that Comply with the ANSI Standard

Use the following 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

The following 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

The following 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));
```

Use of 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

The following 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.

J.D. Edwards Defined Structures

J.D. Edwards software 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 J.D. Edwards software to manipulate them. Do not access the members of these data types directly.

MATH_NUMERIC Data Type

The MATH_NUMERIC data type is commonly used to represent numeric values in J.D. Edwards 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;
```

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

See Also

- *Messaging* in the *Development Tools Guide* for information about creating error messages and workflow messages

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))
```

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 the following tables.

The JDB errors are as follows:

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
078K	Set Sequence of Table Failed
078S *	Initialization of Behavior Failed

* 078S does not use text substitution

The JDE Cache errors are as follows:

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 in lpDS for Error Messages

Include the parameters cSuppressErrorMessage and szErrorMessageID in lpDS for error message processing. The functionality for each is described below:

- 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 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

The following 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;
```

Standard Business Function Error Conditions

Business functions that use the J.D. Edwards 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

The following 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 J.D. Edwards 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 your 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

The following 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);
}
```

DSDE0022 Data Structure

The data structure DSDE0022 contains the single item, szDescription[41]. Use the DSDE0022 data structure for JDB errors and JDE cache errors as the last parameter in jdeErrorSet.

Mapping Data Structure Errors with jdeCallObject

Any Business Function calling another Business Function is required to use jdeCallObject. When using jdeCallObject, be sure to match the Error IDs correctly.

The programmer needs 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);
```

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 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.

The following 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)

Example: Custom Data Dictionary Trigger

An example of a custom data dictionary trigger is the edit trigger lsColumnInAddressBook. This trigger verifies that an address book record exists for the address number passed in szBehaviorEditString and, if it does, returns the alpha name in szDescription001. The variable member for this trigger is mnAddressNumber, which was created using the AN8 data dictionary item.

The C program for this trigger uses the following structure:

```
typedef struct tagDSD0100039

{

   ID                         idBhvrErrorId;

   JCHAR                      szBehaviorEditString[51];

   JCHAR                      szDescription001[31];

   MATH_NUMERIC               mnAddressNumber;

} DSD0100039,FAR *LPDSD0100039;
```

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. This means that normal string functions, such as strlen() and strcpy, do not work with Unicode data.

Do not use the datatype “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
‘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 as follows:

`jdeSxxxxxx()` indicates a Unicode string function

`jdeZSxxxx()` indicates a non-Unicode string function

Some of the replacement functions include the following:

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));`".

See Also

- The *Unicode Codebook* for a complete listing of Unicode string functions

Example: Using Unicode String Functions

The following 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

The following 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 below, 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;
  }
```

The following 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 below 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 the following 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 J.D. Edwards 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. The following 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. The following 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. The following 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 J.D. Edwards ERP 9.0. Use the following 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 J.D. Edwards ERP 9.0.

Example: Third-Party API

The following 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
 * OneWorld® */
jdeToUnicode(szStateName, zzStateName, DIM(szStateName), NULL);
```

Flat-File APIs

ERP 9.0 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 ERP 9.0-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 the following file I/O functionalities: 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 ERP 9.0 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 ERP 9.0, use the non-converting APIs (for example, jdeFprintf()).

Example: Flat-File APIs

The following example writes text to a flat file that would only be read by ERP 9.0. 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);
```

The following 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);
```

Standard Header and Source Files

Business Function Design generates the .c and .h templates. The following sections detail what information goes into each section of the templates and refer you to related information.

Model source code files exist for both the .c and .h modules of your business function.

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.

The following 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 06/06/2003                     - Created
 *
 * Copyright (c) J.D. Edwards World Source Company, 2003
 *
 * This unpublished material is proprietary to J.D. Edwards World
 * Source Company.  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 J.D. Edwards World Source Company.
 *************************************************************/


#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 J.D. Edwards & Company copyright notice and must be included in each source file. Do not change this section.

Header Definition for a Business Function

The Header Definition for a Business Function section 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 includes the table headers associated with tables directly accessed by the business function.

See Also

- *Lowercase Letters in Include Statements* in the *Development Standards Business Function Programming Guide*

External Business Function Header Inclusions

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

See Also

- *Lowercase Letters in Include Statements* in the *Development Standards Business Function Programming Guide*

Global Definitions

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

See Also

- *Using Define Statements* in the *Development Standards Business Function Programming Guide*

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 Also

- *Naming Conventions* in the *Development Standards Business Function Programming Guide*
- *Using Typedef Statements* in the *Development Standards Business Function Programming Guide*

DS Template Type Definitions

Do not modify the DS Template Type Definitions section. The DS Template Type Definitions section defines the business functions contained in the source that correspond to the header. The structure is generated from the business function or data structure design window in Object Management Workbench.

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.

See Also

- *Creating Function Prototypes* in the *Development Standards Business Function Programming Guide*

Internal Function Prototypes

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

See Also

- *Naming Conventions* in the *Development Standards Business Function Programming Guide*
- *Creating Function Prototypes* in the *Development Standards Business Function Programming Guide*

Standard Source

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

A template generated for a standard source file when you create a J.D. Edwards software 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 06/06/1997                     - Created
 *
 * Copyright (c) J.D. Edwards World Source Company, 1996
 *
 * This unpublished material is proprietary to J.D.Edwards World
 * SourceCompany.
 * 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
 * J.D. Edwards World Source Company.
 *************************************************************/
/*************************************************************
```

```
 * 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 J.D. Edwards & Company 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.

See Also

- *Using Define Statements* in the *Development Standards Business Function Programming Guide*

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 Also

- *Naming Conventions* in the *Development Standards Business Function Programming Guide*
- *Initializing Variables* in the *Development Standards Business Function Programming Guide*

Declare Structures

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

See Also

- *Variable Names* in the *Development Standards Business Function Programming Guide*

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.

See Also

- *Variable Names* in the *Development Standards Business Function Programming Guide*

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.

See Also

- *Declaring and Initializing Variables and Data Structures* in the *Development Standards Business Function Programming Guide*

Main Processing

Use the Main Processing section to write your code.

Function Clean Up

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

See Also

- *Using the Function Clean Up Area* in the *Development Standards Business Function Programming Guide*

Internal Function Comment Block

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

See Also

- *Naming Conventions* in the *Development Standards Business Function Programming Guide*
- *Creating Function Prototypes* in the *Development Standards Business Function Programming Guide*