JD Edwards EnterpriseOne Tools

APIs and Business Functions Guide

Release 9.1

E24234-01

December 2011

ORACLE

JD Edwards EnterpriseOne Tools APIs and Business Functions Guide, Release 9.1

E24234-01

Copyright © 2011, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

3 Using Business Functions

Preface

Welcome to the JD Edwards EnterpriseOne Tools Development Tools: APIs and Business Functions Guide.

Audience

This guide is intended for developers and technical consultants who are responsible for working with APIs and business functions.

This guide assumes you have a working knowledge of the following:

- C++ programming language
- JD Edwards EnterpriseOne event rules

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

You can access related documents from the JD Edwards EnterpriseOne Release Documentation Overview pages on My Oracle Support. Access the main documentation overview page by searching for the document ID, which is 876932.1, or by using this link:

https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=876932.1

To navigate to this page from the My Oracle Support home page, click the Knowledge tab, and then click the Tools and Training menu, JD Edwards EnterpriseOne, Welcome Center, Release Information Overview.

This guide contains references to server configuration settings that JD Edwards EnterpriseOne stores in configuration files (such as jde.ini, jas.ini, jdbj.ini, jdelog.properties, and so on). Beginning with the JD Edwards EnterpriseOne Tools Release 8.97, it is highly recommended that you only access and manage these settings

for the supported server types using the Server Manager program. See the Server Manager Guide.

Conventions

The following text conventions are used in this document:

Convention	Meaning
Bold	Indicates field values.
Italics	Indicates emphasis and JD Edwards EnterpriseOne or other book-length publication titles.
`Monospace`	Indicates a JD Edwards EnterpriseOne program, other code example, or URL.

1

Introduction to JD Edwards EnterpriseOne Tools: APIs and Business Functions

This chapter contains the following topics:

1.1 APIs and Business Functions Overview

JD Edwards EnterpriseOne Tools APIs and business functions are used to create complex, reusable routines in C. Business functions can call APIs directly and can in turn be invoked from event rules (ER).

1.2 APIs and Business Functions Implementation

The following implementations steps need to be performed before working with JD Edwards EnterpriseOne Tools APIs and business functions:

1. Configure Object Management Workbench.

   See "Configuring JD Edwards EnterpriseOne OMW" in the *JD Edwards EnterpriseOne Tools Object Management Workbench Guide*.

2. Configure Object Management Workbench user roles and allowed actions.

   See "Configuring User Roles and Allowed Actions" in the *JD Edwards EnterpriseOne Tools Object Management Workbench Guide*.

3. Configure Object Management Workbench functions.

   See "Configuring JD Edwards EnterpriseOne OMW Functions" in the *JD Edwards EnterpriseOne Tools Object Management Workbench Guide*.

4. Configure Object Management Workbench activity rules.

   See "Configuring Activity Rules" in the *JD Edwards EnterpriseOne Tools Object Management Workbench Guide*.

5. Configure Object Management Workbench save locations.

   See "Configuring Object Save Locations" in the *JD Edwards EnterpriseOne Tools Object Management Workbench Guide*.

6. Set up default location and printers.

   See *JD Edwards EnterpriseOne Tools Report Printing Administration Technologies Guide*.

2

Working with APIs

This chapter contains the following topics:

2.1 Understanding APIs

This section discusses:

- API fundamentals
- Common library APIs
- Database APIs

2.1.1 API Fundamentals

APIs are routines that perform predefined tasks. JD Edwards EnterpriseOne APIs make it easier for third-party applications to interact with JD Edwards EnterpriseOne software. These APIs are functions that you can use to manipulate JD Edwards EnterpriseOne data types, provide common functionality, and access the database. Several categories of APIs exist, including the Common Library Routines and JD Edwards EnterpriseOne Database (JDEBASE) APIs.

Programing with APIs is useful for these reasons:

- No code modifications are required as functionality is upgraded.
- When a data structure changes, source modifications are minimal to nonexistent.
- Common functionality is provided through the APIs, and they are less prone to error.

When the code in an API changes, business functions typically only need to be recompiled and relinked.

2.1.2 Common Library APIs

The Common Library APIs, such as determining whether foreign currency is enabled, manipulating the date format, retrieving link list information, or retrieving math numeric and date information are specific to JD Edwards EnterpriseOne functionality.

You can use these APIs to set up data by calling APIs and modifying data after API calls. Some of the more commonly used categories of APIs include MATH_NUMERIC, JDEDATE, and LINKLIST. Other miscellaneous Common Library APIs are also available.

JD Edwards EnterpriseOne provides the data types, MATH_NUMERIC and JDEDATE, for use when creating business functions. Because these data types might change, you must use the Common Library APIs provided by JD Edwards EnterpriseOne to manipulate the variables of these data types.

2.1.2.1 MATH_NUMERIC Data Type

The MATH_NUMERIC data type exclusively represents all numeric values in JD Edwards EnterpriseOne software. The values of all numeric fields on a form or batch process are communicated to business functions in the form of pointers to MATH_ NUMERIC data structures. MATH_NUMERIC is used as a data dictionary (DD) data type.

The data type is defined as follows:

```
struct tagMATH_NUMERIC
{
 ZCHAR String[MAXLEN_MATH_NUMERIC+1];/* Just the digits - no separators */
 BYTE Sign;    /* - if negative, 0x00 otherwise */
 ZCHAR EditCode;   /* The Data Dictionary edit code to Format for display */
 short nDecimalPosition;  /* # of digits from right end of string to decimal
point⇒
 */
 short nLength;   /* The number of digits in s */
 WORD wFlags;   /* Processing Flags */
 ZCHAR szCurrency[CURRENCY_CODE_SIZE];/* The Currency Code */
 short nCurrencyDecimals;  /* The Number of Currency Decimals */
 short nPrecision;   /* The Data Dictionary Size */
};
```

This table lists various elements:

MATH_NUMERIC Element	Description
String	Digits without separators
Sign	A minus sign indicates the number is negative, otherwise the value is 0x00
EditCode	Data dictionary edit code that formats the number for display
nDecimalPosition	Number of digits from the right to place the decimal
nLength	Number of digits in the string
wFlags	Processing flags
szCurrency	Currency code
nCurrencyDecimals	Number of currency decimals
nPrecision	Data dictionary size

2.1.2.2 JDEDATE Data Type

The JDEDATE data type exclusively represents all dates in JD Edwards EnterpriseOne software. The values of all date fields on a form or batch process are communicated to business functions in the form of pointers to JDEDATE data structures. JDEDATE is used as a data dictionary data type.

This code sample illustrates defining the data type:

```
struct tagJDEDATE
{
 short nYear;;
 short nMonth;;
 short nDay;
};
typedef struct tagJDEDATE JDEDATE, FAR *LPJDEDATE;
```

This table lists the elements in the JDEDATE data type:

JDEDATE Element	Description
nYear	Year (4 digits)
nMonth	Month
nDay	Day

2.1.3 Database APIs

JD Edwards EnterpriseOne software supports multiple databases. An application can access data from a number of databases.

2.1.3.1 Standards and Portability

These standards affect the development of relational databases:

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

Ideally, industry standards enable users to work identically with different relational database systems. Although each major vendor supports industry standards, it also offers extensions to enhance the functionality of the SQL language. Vendors also periodically release upgrades and new versions of their products.

These extensions and upgrades affect portability. Due to the industry impact of software development, applications need a standard interface to databases that is not affected by differences between database vendors. When a vendor provides a new release, the affect on existing applications should be minimal. To solve many of these portability issues, many organizations use standard database interfaces called open database connectivity (ODBC).

2.1.3.2 JD Edwards EnterpriseOne ODBC

JD Edwards EnterpriseOne ODBC enables you to use one set of functions to access multiple relational database management systems. Consequently, you can develop and compile applications knowing that they can run on a variety of database types with the correct database driver. Database drivers are installed that enable the JD Edwards EnterpriseOne ODBC interface to communicate with a specific database system using a database driver.

The driver handles the I/O buffers to the database, which enables a programmer to write an application that communicates with a generic data source. The database driver is responsible for processing the API request and communicating with the correct data source. The application does not have to be recompiled to work with other databases. If the application must perform the same operation with another database, a new driver is loaded.

A driver manager handles all application requests to the JD Edwards EnterpriseOne database function call. The driver manager processes the request or passes it to an appropriate driver.

JD Edwards EnterpriseOne applications access data from heterogeneous databases, using the JDB API to interface between the applications and multiple databases. Applications and business functions use the JDB API to dynamically generate platform-specific SQL statements. JDB also supports additional features, such as replication and cross-data source joins.

2.1.3.3 Standard JDEBASE API Categories

You can use control and request level APIs to develop and test business functions. This table lists the categories of JDEBASE APIs:

Category	Description
Control Level	Provides functions for initializing and terminating the database connection.
Request Level	Provides functions for performing database transactions. The request level functions perform these tasks: ■ Connect to and disconnect from tables and business views in the database. ■ Perform data manipulation operations of select, insert, update, and delete. ■ Retrieve data with fetch commands.
Column Level	Performs and modifies information for columns and tables.
Global Table/Column Specifications	Provides the capability to create and manipulate column specifications.

2.1.3.4 Connecting to a Database

To perform a request, the driver manager and driver must manage the information for the development environment, each application connection, and the SQL statement. The pointers that return this information to the application are called handles. The APIs must include these handles in each function call. Handles used by the development environment include these handles:

Handle	Purpose
HENV	The environment handle contains information related to the current database connection and valid connection handles. Every application connecting to the database must have an environment handle. This handle is required to connect to a data source.
HUSER	The user handle contains information related to a specific connection. Each user handle has an associated environment handle with it. A connection handle is required to connect to a data source. If you are using transaction processing, initializing HUSER indicates the beginning of a transaction.
HREQUEST	The request handle contains information related to a specific request to a data source. An application must have a request handle before executing SQL statements. Each request handle is associated with a user handle.

2.1.3.5 Understanding Database Communication Steps

Several APIs called in succession can perform these steps for database communication:

- Initialize communication with the database.
- Establish a connection to the specific data to access.
- Execute statements on the database.
- Release the connection to the database.
- Terminate communication with the database.

This table lists some of the API levels and the communication handles and API names that are associated with them:

API Level	Communication Handles	API Name
Control level (application or test driver)	Environment handle	**JDB_InitEnv**
Control level (application or test driver)	User handle (created)	**JDB_InitUser**
Request level (business function)	User handle (retrieved)	**JDB_InitBhvr**
Request level (business function)	Request handle	**JDB_OpenTable**
Request level (business function)	Request handle	**JDB_FetchKeyed()**
Request level (business function)	Request handle	**JDB_CloseTable**
Request level (business function)	User handle	**JDB_FreeBhvr**
Control level (application or test driver)	User handle	**JDB_FreeUser**
Control level (application or test driver)	Environment handle	**JDB_FreeEnv**

2.2 Calling APIs

This section discusses how to:

- Call an API from an external business function.
- Call a Visual Basic program from JD Edwards EnterpriseOne software.

2.2.1 Calling an API from an External Business Function

You can call APIs from external business functions. To call an API from an external business function, you must first determine the function-calling convention of the .dll that you are going to use. It can be either cdecl or stdcall. The code might change slightly depending on the calling convention. This information should be included in the documentation for the .dll. If you do not know the calling convention of the .dll, you can execute the *dumpbin* command to determine the calling convention. Execute this command from the MSDOS prompt window:

```
dumpbin /EXPORTS ExternalDll.DLL.
```

Dumpbin displays information about the dll. If the output contains function names preceded by _ and followed by an @ sign with additional digits, the dll uses the stdcall calling convention; otherwise, it uses cdecl.

2.2.1.1 Stdcall Calling Convention

This example is standard code for Windows programs and is not specific to JD Edwards EnterpriseOne software:

```
# ifdef JDENV_PC
HINSTANCE hLibrary = LoadLibrary(_TEXT(YOUR_LIBRARY.DLL)); // substitute the name⇒
 of the external dll
if(hLibrary)
{
// create a typedef for the function pointer based on the parameters and return⇒
 type of the function to be called. This information can be obtained
// from the header file of the external dll. The name of the function to be
called⇒
 in the following code is StartInstallEngine. We create a typedef for
// a function pointer named PFNSTARTINSTALLENGINE. Its return type is BOOL. Its⇒
 parameters are HUSER, LPCTSTR, LPCTSTR, LPTSTR & LPTSTR.
// Substitute these with parameter and return types for the particular API.
typedef BOOL (*PFNSTARTINSTALLENGINE) (HUSER, LPCTSTR, LPCTSTR, LPTSTR, LPTSTR);
// Now create a variable for the function pointer of the type you just created.⇒
 Then make call to GetProcAddress function with the first
// parameter as the handle to the library you just loaded. The second parameter⇒
 should be the name of the function you want to call prepended
// with an _, and appended with an @ followed by the total number of bytes for
the⇒
 parameters. In this example, the total number of bytes in the
// parameters for StartInstallEngine is 20 ( 4 bytes for each parameter ). The
Get⇒
ProcAddress API will return a pointer to the function that you need to
// call.
PFNSTARTINSTALLENGINE lpfnStartInstallEngine = (PFNSTARTINSTALLENGINE) GetProc⇒
Address(hLibrary, _StartInstallEngine@20);
if ( lpfnStartInstallEngine )
{
// Now call the API by passing in the requisite parameters.
lpfnStartInstallEngine(hUser, szObjectName, szVersionName, pszObjectText,
szObject⇒
Type);
}
#endif
```

2.2.1.2 Cdecl Calling Convention

The process for using the cdecl calling convention is similar to the process for using the std calling convention. They differ principally in the second parameter for **GetProcAddress**. Note the comments that precede that call.

```
# ifdef JDENV_PC
HINSTANCE hLibrary = LoadLibrary(_TEXT(YOUR_LIBRARY.DLL)); // substitute the name⇒
 of the external dll
if(hLibrary)
{
// create a typedef for the function pointer based on the parameters and return⇒
 type of the function to be called. This information can be obtained
// from the header file of the external dll. The name of the function to be
called⇒
 in the following code is StartInstallEngine. We create a typedef for
// a function pointer named PFNSTARTINSTALLENGINE. Its return type is BOOL. Its⇒
 parameters are HUSER, LPCTSTR, LPCTSTR, LPTSTR & LPTSTR.
// Substitute these with parameter and return types for the particular API.
typedef BOOL (*PFNSTARTINSTALLENGINE) (HUSER, LPCTSTR, LPCTSTR, LPTSTR, LPTSTR);
```

```
// Now create a variable for the function pointer of the type you just created.⇒
 Then make call to GetProcAddress function with the first
// parameter as the handle to the library you just loaded. The second parameter⇒
 should be the name of the function you want to call. In this
// case it will be StartInstallEngine only. The GetProcAddress API will return a⇒
 pointer to the function that you need to call.
PFNSTARTINSTALLENGINE lpfnStartInstallEngine = (PFNSTARTINSTALLENGINE) GetProc⇒
Address(hLibrary, StartInstallEngine);
if ( lpfnStartInstallEngine )
{
// Now call the API by passing in the requisite parameters.
lpfnStartInstallEngine(hUser, szObjectName, szVersionName, pszObjectText,
szObject⇒
Type);
}
#endif
```

Note: These calls work only on a Windows *client* machine. **LoadLibrary** and **GetProcAddress** are Windows APIs. If the business function is compiled on a *server*, the compile will fail.

2.2.2 Calling a Visual Basic Program from JD Edwards EnterpriseOne Software

You can call a Visual Basic program from a JD Edwards EnterpriseOne business function and pass a parameter from the Visual Basic program to the JD Edwards EnterpriseOne business function using this process:

1. Write the Visual Basic program into a Visual Basic .dll that exports the function name of the program and returns a parameter to the JD Edwards EnterpriseOne business function.
2. Write a business function that loads the Visual Basic .dll using the win32 function LoadLibrary.
3. In the business function that you create, call the win32 function GetProcAddress to get the Visual Basic function and call it.

2.3 Using the SAX Parser

This section provides an overview of the SAX parser and of examples for its use.

2.3.1 Understanding the SAX Parser

The SAX parser is one of two main parsers used for XML data. It is an events-based parser, as opposed to the other XML parser, DOM, which is a tree-based parser. The Xerces product, from the Apache organization, provides both XML parsers. The Xerces code is written in C++. To make XML parsing available to business functions, a C-API interface, XercesWrapper, exists to provide access to both parsers. The design of the parsers is quite different, and that provides advantages for each parser, depending on the intended usage.

The DOM parser reads the XML file and builds an internal model (DOM document tree) of that file in memory. This has the advantage of enabling you to traverse the tree, retrieve parent-child relationships, and revisit the same data multiple times. The disadvantages include high memory requirements for large XML files. Also, the entire XML file must be read into memory before any of the data in the DOM document tree can begin to be processed. The DOM parser can also be used to programmatically

build a DOM document tree in memory, and then write that tree to a file, in XML format.

The SAX parser reads an XML file and as each item is read, the parser passes that piece of data to callback functions. This methodology has the advantage of enabling fast processing with minimal memory usage. Also, the parsing can be stopped after a specific item has been found. The disadvantages include that the current state of parsing must be maintained by the callback functions, and previous data items can not be revisited without rereading the XML file. Finally, the SAX parser is a read-only parser.

This is a typical sequence used for parsing an XML data file using the DOM parser:

1. Initialize the XercesWrapper, which in turn, initializes the Xerces code.
2. Initialize the DOM parser.
3. Parse the XML data file.
4. Retrieve a pointer to the root element of the DOM document tree.
5. Retrieve additional elements and data, by traversing the DOM document tree.

   The callback functions are called whenever the specified events in the XML file are parsed.
6. Free all DOM elements that have been retrieved.
7. Free the DOM document tree.
8. Free the DOM parser.
9. Terminate the XercesWrapper interface, which in turn, closes the Xerces code.

This is a typical sequence used for parsing an XML data file, using the SAX parser:

1. Initialize the XercesWrapper, which in turn, initializes the Xerces code.
2. Initialize the SAX parser.
3. Set up various callback functions for specific parsing events.
4. Parse the XML data file.
5. Call the callback functions as each event in the XML file is parsed.
6. Within the callback functions, process the retrieved data and maintain a context for coordination between callback functions.
7. Free the SAX parser.
8. Terminate the XercesWrapper interface, which in turn, closes the Xerces code.

2.3.2 Examples of SAX Parser Usage

Many of the initialization, parsing, and termination functions are the same for both SAX and DOM parsers. The major difference is that the DOM parser returns a document handle which is then used with the traversing and data retrieval functions. Those functions are not used with SAX. SAX does all of the data processing within the user-defined callback functions. The callback functions are not used with DOM.

The processing of SAX-parsed data items occurs within the callback functions. Typically, each callback function maintains a context. The context can be passed to all callback functions and can be implemented as a data structure. The context, plus the other data passed to the callback functions, enables each data item to be processed appropriately.

2.3.2.1 Example Context Data Structure

This is a sample function which uses the SAX parser:

```
typedef struct tagParserCallbackValues {
 FILE *fp;
 JCHAR *szIndentString;
 int nIndentLevel;
} ZCALLBACK_VALUES, *PCALLBACK_VALUES;
```

2.3.2.2 Example Main Function

This is a sample context data structure:

```
/* SAX callbacks - display callback events into file */
int testcase_read_15(JCHAR *m_infile, JCHAR *m_outfile)
{
 XRCS_Status XRCSStatus;
 XRCS_hParser hParser;
 ZCALLBACK_VALUES zCbValues;
 PCALLBACK_VALUES pCbValues = &zCbValues;

 /* initialize context structure */
 pCbValues->fp = NULL;
 pCbValues->szIndentString = _J(" ");
 pCbValues->nIndentLevel = 0;

 /* open display file */
 pCbValues->fp = jdeFopen(m_outfile, _J("w"));

 if (pCbValues->fp != NULL)
 {
  XRCSStatus = XRCS_initEngine();
  if(XRCSStatus != XRCS_SUCCESS) {
   return -1;
  }

  XRCSStatus = XRCS_getParserByType(&hParser, XRCS_SAX_PARSER_TYPE);
  if(XRCSStatus != XRCS_SUCCESS) {
   return -1;
  }

  XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_START_DOC,
   (void *) cb_startDoc_Display, (void *) pCbValues);
  if(XRCSStatus != XRCS_SUCCESS) {
   return -1;
  }

  /* set up callbacks for the SAX parser */
  XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_END_DOC,
   (void *) cb_endDoc_Display, (void *) pCbValues);
  if(XRCSStatus != XRCS_SUCCESS) {
   return -1;
  }

  XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_START_ELEM,
   (void *) cb_startElement_Display, (void *) pCbValues);
  if(XRCSStatus != XRCS_SUCCESS) {
   return -1;
  }
```

```
 XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_END_ELEM,
  (void *) cb_endElement_Display, (void *) pCbValues);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_CHARACTERS,
  (void *) cb_characters_Display, (void *) pCbValues);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 XRCSStatus = XRCS_setCallback(hParser,
  XRCS_CALLBACK_IGNORABLE_WHITESPACE,
  (void *) cb_ignorableWhitespace_Display, (void *) pCbValues);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_FATAL_ERROR,
  (void *) cb_fatalError_Display, (void *) pCbValues);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_ERROR,
  (void *) cb_error_Display, (void *) pCbValues);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 XRCSStatus = XRCS_setCallback(hParser, XRCS_CALLBACK_WARNING,
  (void *) cb_warning_Display, (void *) pCbValues);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 /* now do the actual parsing */
 XRCSStatus = XRCS_parseXMLFile(hParser,m_infile, NULL);
 if(XRCSStatus != XRCS_SUCCESS) {
  return -1;
 }

 XRCSStatus = XRCS_freeParser(hParser);
 XRCSStatus = XRCS_terminateEngine();

 /* close display file */
 jdeFclose(pCbValues->fp);
}
else
{
 /* could not open display file */
 return -1; }

return 0;
}
```

2.3.2.3 Example Callback Functions

These are sample callback functions:

```
/* callbacks for display of SAX parser events */
XRCS_CallbackStatus cb_startDoc_Display(void *pContext)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("START DOCUMENT"));
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_endDoc_Display(void *pContext)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("END DOCUMENT"));
 indentNewLine(pCbValues);
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_startElement_Display(void *pContext,
 const JCHAR *szUri,
 const JCHAR *szLocalname,
 const JCHAR *szQname,
 unsigned int nNumAttrs,
 const XRCS_ATTR_INFO *pAttributes)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;
 unsigned int nAttrNum;
 const XRCS_ATTR_INFO * thisAttr = NULL;

 pCbValues->nIndentLevel++;
 /* display element name */
 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("ELEMENT: "));
 if (jdeStrlen( szLocalname) != 0)
 {
  jdeFprintf(pCbValues->fp, _J("<%ls"), szLocalname);
 }
 else
 {
  jdeFprintf(pCbValues->fp, _J("<%ls"), szQname);
 }
 /* display attributes */
 if (nNumAttrs > 0U)
 {
  for (nAttrNum = 0U; nAttrNum < nNumAttrs; nAttrNum++)
  {
   thisAttr = &pAttributes[nAttrNum];
   /* display attrribute name */
   indentNewLine(pCbValues);
   jdeFprintf(pCbValues->fp, _J(" ATTR: "));
   if (jdeStrlen( thisAttr->szAttrLocalname) != 0)
   {
   jdeFprintf(pCbValues->fp, _J("%ls"),
    thisAttr->szAttrLocalname);
   }
```

```
   else
   {
   jdeFprintf(pCbValues->fp, _J("%ls"), thisAttr->szAttrQname);
   }
   /* display attribute value */
   jdeFprintf(pCbValues->fp, _J(" \""));
   jdeFprintf(pCbValues->fp, _J("%ls"), thisAttr->szAttrValue);
   jdeFprintf(pCbValues->fp, _J("\""));
  }
  indentNewLine(pCbValues);
 }
 /* display close of element name */
 jdeFprintf(pCbValues->fp, _J(">"));
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_endElement_Display_Terminate(void *pContext,
 const JCHAR *szUri,
 const JCHAR *szLocalname,
 const JCHAR *szQname)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("END_ELM: "));
 if (jdeStrlen( szLocalname) != 0)
 {
  jdeFprintf(pCbValues->fp, _J("</%ls>"), szLocalname);
 }
 else
 {
  jdeFprintf(pCbValues->fp, _J("</%ls>"), szQname);
 }
 pCbValues->nIndentLevel--;
 return( XRCS_CB_TERMINATE);
}

XRCS_CallbackStatus cb_endElement_Display(void *pContext,
 const JCHAR *szUri,
 const JCHAR *szLocalname,
 const JCHAR *szQname)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("END_ELM: "));
 if (jdeStrlen( szLocalname) != 0)
 {
  jdeFprintf(pCbValues->fp, _J("</%ls>"), szLocalname);
 }
 else
 {
  jdeFprintf(pCbValues->fp, _J("</%ls>"), szQname);
 }
 pCbValues->nIndentLevel--;
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_warning_Display(void *pContext,
 XRCS_CallbackType eCallbackType,
```

```
 int nLineNum,
 int nColNum,
 const JCHAR *szPublicId,
 const JCHAR *szSystemId,
 const JCHAR *szMessage)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("Warning: "));
 jdeFprintf(pCbValues->fp, _J(" %ls (%ls) - %ls found at Column %d
 Line %d"), szSystemId, szPublicId, szMessage, nColNum, nLineNum);
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_error_Display(void *pContext,
 XRCS_CallbackType eCallbackType,
 int nLineNum,
 int nColNum,
 const JCHAR *szPublicId,
 const JCHAR *szSystemId,
 const JCHAR *szMessage)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("Error: "));
 jdeFprintf(pCbValues->fp, _J(" %ls (%ls) - %ls found at Column %d
 Line %d"), szSystemId, szPublicId, szMessage, nColNum, nLineNum);
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_fatalError_Display(void *pContext,
 XRCS_CallbackType eCallbackType,
 int nLineNum,
 int nColNum,
 const JCHAR *szPublicId,
 const JCHAR *szSystemId,
 const JCHAR *szMessage)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;

 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("Fatal Error: "));
 jdeFprintf(pCbValues->fp, _J(" %ls (%ls) - %ls found at Column %d Line %d"),
 szSystemId, szPublicId, szMessage, nColNum, nLineNum);
 return( XRCS_CB_TERMINATE);
}

XRCS_CallbackStatus cb_characters_Display(void *pContext,
 const JCHAR *szText)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;
 int nTextLen;
 int nTextRemaining;
 int nTextPieceLen;
 int nTextStartPosition;

 nTextLen = jdeStrlen( szText);
 indentNewLine(pCbValues);
```

```
 jdeFprintf(pCbValues->fp, _J("CHARS: "));
 if (hasPrintingChars( szText, nTextLen) == TRUE)
 {
  /* initial quote */
  jdeFprintf(pCbValues->fp, _J("\""), szText);
  /* actual text, output in blocks of 10000 characters */
  /* jdeFprintf will not work with very large strings */
  nTextRemaining = nTextLen;
  nTextStartPosition = 0;
  while (nTextRemaining > 0)
  {
   if (nTextRemaining > 10000)
   {
   nTextPieceLen = 10000;
   }
   else
   {
   nTextPieceLen = nTextRemaining;
   }
   jdeFprintf(pCbValues->fp, _J("%.*ls"), nTextPieceLen,
   (JCHAR *) &(szText[nTextStartPosition]));
   nTextRemaining -= nTextPieceLen;
   nTextStartPosition += nTextPieceLen;
  }
  /* trailing quote */
  jdeFprintf(pCbValues->fp, _J("\""), szText);
 }
 return( XRCS_CB_CONTINUE);
}

XRCS_CallbackStatus cb_ignorableWhitespace_Display(void *pContext,
 const JCHAR *szText)
{
 PCALLBACK_VALUES pCbValues = (PCALLBACK_VALUES) pContext;
 int nTextLen;

 nTextLen = jdeStrlen( szText);
 indentNewLine(pCbValues);
 jdeFprintf(pCbValues->fp, _J("IGNORABLE WHITESPACE: "));
 if (hasPrintingChars( szText, nTextLen) == TRUE)
 {
  jdeFprintf(pCbValues->fp, _J("\"%ls\""), szText);
 }
 return( XRCS_CB_CONTINUE);
}

void indentNewLine(PCALLBACK_VALUES pCbValues)
{
 int nIndent = 0;

 jdeFprintf(pCbValues->fp,
  _J("\n"));

 while (nIndent < pCbValues->nIndentLevel)
 {
  jdeFprintf(pCbValues->fp, _J("%ls"), pCbValues->szIndentString);
  nIndent++;
 }
}
```

```
BOOL hasPrintingChars( const JCHAR *szText, int nTextLen)
{
 BOOL bHasPrinting = FALSE;
 int nText = 0;

 /* true if contains any printing characters */
 /* false if all blanks or control characters */
 while (nText < nTextLen)
 {
  if (szText[nText] > _J(' '))
  {
   bHasPrinting = TRUE;
   break;
  }
  nText++;
 }
 return( bHasPrinting);
}
```

2.3.3 Example of a SAX Parsing Sequence

This is an example of the sequence of callback functions called, for an example string of XML data. Before parsing, these callback functions were set up:

- **cb_startAllElements** for start-of-element event type.
- **cb_endAllElements** for end-of-element event type.
- **cb_startElement1** for start-of-element, with optional name specified as "elapsedTime."
- **cb_endElement1** for end-of-element, with optional name specified as "elapsedTime."
- **cb_chars** for characters event type.
- **cb_allCharacters** for characters, with optional setting for characters after elements.
- **cb_fatalError** for fatal-error event type.

The example XML string to be parsed is:

```
<main>startMain<elapsedTime>123</elapsedTime>endMain</main>
```

This callback sequence results from parsing this XML string:

- **cb_startAllElements** for *main*.
- **cb_chars** for *startMain*.
- **cb_allCharacters** for *startMain*.
- **cb_startAllElements** for *elapsedTime*.
- **cb_startElement1** for *elapsedTime*.
- **cb_chars** for *123*.
- **cb_allCharacters** for *123*.
- **cb_endAllElements** for *elapsedTime*.
- **cb_endElement1** for *elapsedTime*.
- **cb_allCharacters** for *endMain*.

- **cb_endAllElements** for *main*.
- **cb_fatalError** is not called while parsing this example XML string.

2.4 Working with JDECACHE

This section provides overviews of caching, JDECACHE standards, and the JDECACHE API set, and discusses how to:

- Call JDECACHE APIs.
- Set up indices.
- Initialize the cache.
- Use an index to access the cache.
- Use the jdeCacheInit/jdeCacheTerminate rule.
- Use the same cache in multiple business functions or forms.

2.4.1 Understanding Caching

Caching is a process that stores a local copy of frequently accessed content of remote objects. Caching can improve performance. JD Edwards EnterpriseOne software caches information in these ways:

- The system automatically caches some tables, such as those associated with constants, when it reads them from the database at startup.

  It caches these tables to a user's workstation or to a server for faster data access and retrieval.

- Individual applications can be enabled to use cache.

  JDECACHE APIs enable the server or workstation memory to be used as temporary storage.

JDECACHE is a component of JDEKRNL that can hold any type of indexed data that the application needs to store in memory, regardless of the platform on which the application is running; therefore, an entire table can be read from a database and stored in memory. No limitations exist regarding the type of data, size of data, or number of data caches that an application can have, other than the limitations of the computer on which it is running. Both fixed-length and variable-length records are supported. To use JDECACHE on any supported platform, you need to know only a simple set of API calls.

Data handled by JDECACHE is in RAM. Therefore, ensure that you really need to use JDECACHE. If you use JDECACHE, design the records and indices carefully. Minimize the number of records that you store in JDECACHE because JD Edwards EnterpriseOne software and various other applications need this memory as well.

JDECACHE supports multiple cursors, multiple indexes, and partial keys processing. JDECACHE is flexible in terms of positioning within the cache for data manipulation, which improves performance by reducing searching within the cache.

The JDB environment creates, manages, and destroys the JDECACHE environment. Each cache that you use within the JDECACHE environment is associated with a JDB user. Therefore, you must call JDB_InitBhvr API before you call any of the JDECACHE APIs.

2.4.1.1 When to Use JDECACHE

Here is a scenario that highlights when an application might use the JDECACHE APIs.

You use workfiles when an application must store records that a user enters in a detail area until OK processing is activated upon the *Button Clicked* event. On OK processing, all records must be simultaneously updated to the database. This is similar to transaction processing. For example, in the detail area of purchase order detail, if a user enters 30 lines of information and then decides to cancel the transaction, all records in the workfile are deleted and nothing is written to the database. As the user exits each detail row, editing takes place for each field, and then that record is written to the workfile.

If you implement this situation without using workfiles, irreversible updates to database tables occur when the user exits each row. Using workfiles enables you to limit updates to tables so that they only occur on OK button processing, and they are included in a transaction boundary. The workfile defines a data boundary for the grid for processing purposes. This is useful when multiple applications or processes (such as business functions) must access the data in the workfile for updates and calculations.

Using cache might increase performance in some cases. You can use JDECACHE to store in memory the records that the user enters in one purchase order. The number of records that you store depends on the cache buffer size for each record, the local memory size, the location in which the business function that you use runs (for example, server or workstation), and so on. Typically, you should not store more than 1000 records. For example, do not cache the entire Address Book table in memory.

2.4.1.2 Performance Considerations

Follow these guidelines to get the best JDECACHE performance:

- Cache as few records as possible.
- The fewer columns (segments) that you use, the faster the search, insert, and delete actions occur.

  In some cases, the system might have to compare each column before it determines whether to go further in the cache.
- The fewer records in the cache, the faster all operations proceed.

2.4.2 Understanding the JDECACHE API Set

You use a set of public APIs to interact with JDECACHE. You must understand how the JDECACHE APIs are organized to implement them effectively.

2.4.2.1 JDECACHE Management APIs

You can manage cache using the JDECACHE management APIs for these purposes:

- Setting up the cache.
- Clearing the cache.
- Terminating the cache.

Use the **jdeCacheGetNumRecords** and **jdeCacheGetNumCursors** APIs to retrieve cache statistics. They are only passed the HCACHE handle. All other JDECACHE management APIs should always be passed these handles:

- HUSER

- HCACHE

These two handles are essential for cache identification and cache management.

The set of JDECACHE management APIs consist of these APIs:

- **jdeCacheInit**
- **jdeCacheInitEx**
- **jdeCacheInitMultipleIndex**
- **jdeCacheInitMultipleIndexEx**
- **jdeCacheInitUser**
- **jdeCacheInitMultipleIndexUser**
- **jdeCacheGetNumRecords**
- **jdeCacheGetNumCursors**
- **jdeCacheClear**
- **jdeCacheTerminate**
- **jdeCacheTerminateAll**

The **jdeCacheInit** and **jdeCacheInitMultipleIndex** APIs initialize the cache uniquely per user. Therefore, if a user logs in to the software and then runs two sessions of the same application simultaneously, the two application sessions will share the same cache. Consequently, if the first application deletes a record from the cache, the second application cannot access the record. Conversely, if two users log in to the software and then run the same application simultaneously, the two application sessions have different caches. Consequently, if the first application deletes a record from its cache, the second application will still be able to access the record in its own cache. The **jdeCacheInitEx** and **jdeCacheInitMultipleIndexEx** APIs function exactly the same, but they additionally enable you to define the maximum number of cursors that can be opened by the cache.

The **jdeCacheInitUser** and **jdeCacheInitMultipleIndexUser** APIs initialize the cache uniquely per application. Therefore, if a user logs in to the software and then runs two sessions of the same application simultaneously, the two application sessions will have different caches. Consequently, if the first application deletes a record from its cache, the second application can still access the record in its own cache.

2.4.2.2 JDECACHE Manipulation APIs

You can use the JDECACHE manipulation APIs for retrieving and manipulating the data in the cache. Each API implements a cursor that acts as pointer to a record that is currently being manipulated. This cursor is essential for navigation within the cache. JDECACHE manipulation APIs should be passed handles of these types:

- HCACHE

  Identifies the cache that is being worked.

- HJDECURSOR

  Identifies the position in the cache that is being worked.

The set of JDECACHE manipulation APIs contain these APIs:

- **jdeCacheOpenCursor**
- **jdeCacheResetCursor**

- **jdeCacheAdd**
- **jdeCacheFetch**
- **jdeCacheFetchPosition**
- **jdeCacheUpdate**
- **jdeCacheDelete**
- **jdeCacheDeleteAll**
- **jdeCacheCloseCursor**
- **jdeCacheFetchPositionByRef**
- **jdeCacheSetIndex**
- **jdeCacheGetIndex**

2.4.3 Understanding JDECACHE Standards

It is recommended that you apply several standards when using JDECACHE. This section discusses the standards for business functions and programming.

The cache business function name should follow the standard naming convention for business functions.

2.4.3.1 Cache Business Function Source Description

These standards apply to source descriptions for cache business functions:

- The cache business function description must follow the business function description standards.
- The first word must be the noun, *Cache*.
- The second word must be the verb, *Process*.
- For an individual cache function, the words following *Process* should describe the cache. For a common cache function, the words following *Process* should describe the group to which the individual cache functions belong.

These standards apply to cache business function descriptions:

- If the source file contains an individual function, the function name must match the source name.
- If the source file contains a group of cache functions, the individual function names must follow the same standards as the Cache Business Function Source Description standards.

2.4.3.2 Cache Programming Standards

A variety of cache programming standards apply:

- General standards.
- Cache termination instead of clearing.
- Cache name.
- Cache data structure definition.
- Data structure standard data items.
- Cache action code standards.

- Group cache business function header file.
- Individual cache business function header file.

2.4.4 Prerequisites

Before you can use JDECACHE, you must:

- Define an index

  The index specifies to the cache the fields in a record that are used to uniquely identify a cache record.

- Initialize a cache

  Each group of data that an index references requires a separate cache.

2.4.5 Calling JDECACHE APIs

JDECHACHE APIs must be called in a certain order. This list defines the order in which the JDECACHE-related APIs must be called:

1. Call **JDB_InitBhvr**.
2. Create index or indices.
3. Call **jdeCacheInit**, **jdeCacheInitEx**, **jdeCacheInitMultipleIndex**, or **jdeCacheInitMultipleIndexEx**.
4. Call **jdeCacheAdd**.
5. Call **jdeCacheOpenCursor**.
6. Call JDECACHE Operations.

At JDECACHE Operations, the actual JDECACHE APIs can be called in any order. The operations in this list of JDECACHE operations can occur in any order:

- **jdeCacheFetch**
- **jdeCacheOpenCursor** (the second cursor)
- **jdeCacheFetchPosition**
- **jdeCacheUpdate**
- **jdeCacheDelete**
- **jdeCacheDeleteAll**
- **jdeCacheResetCursor**
- **jdeCacheCloseCursor** (if the second cursor is opened)
- **jdeCacheCloseCursor**
- **jdeCacheTerminate**
- **JDB_FreeBhvr**

2.4.6 Setting Up Indexes

To store or retrieve any data in JDECACHE, you must set up at least one index that consists of at least one column. The index is limited to a maximum of 25 columns (which are called segments) in the index structure. Use the data type provided to tell the cache manager what the index looks like. You must provide the number of

columns (segments) in the index and the offset and size of each column in the data structure. To maximize performance, minimize the number of segments.

This code is the definition of the structure that holds index information:

```
#define JDECM_MAX_UM_SEGMENTS 25
struct _JDECMKeySegment
{
 short int nOffset;  /* Offset from beginning of structure in bytes */
 short int nSize;  /* Size of data item in bytes */
 int idDataType;  /* EVDT_MATH_NUMERIC or EVDT_STRING*/
} JDECMKEYSEGMENT;
struct _JDECMKeyStruct
{
 short int nNumSegments;
 JDECMKEYSEGMENT CacheKey[JDECM_MAX_NUM_SEGMENTS];
} JDECMINDEXSTRUCT;
```

Observe these rules when you create indices in JDECACHE:

- Always declare the index structure as an array that holds one element for single indexes.

  Declare the index structure as an array that holds more than one element for multiple indexes. You can create an unlimited number of indexes.

- Always use `memset()` for the index structure.

  When you use `memset()` for multiple indexes, multiply the size of the index structure by the total number of indexes.

- Always assign as elements the number of segments that correspond to the number of columns that you have in the CacheKey array.

- Always use `offsetof ()` to indicate the offset of a column in the structure that contains the columns.

This example illustrates a single index with multiple fields:

```
/* Example of single index with multiple fields.*/
JDECMINDEXSTRUCT Index[1]    = {0};
memset(&dsCache,0x00,sizeof(dsCache));
/* Initialize cache. */
Index->nNumSegments=5;
Index->CacheKey[0].nOffset=offsetof(DSCACHE,szEdiUserId);
Index->CacheKey[0].nSize=DIM(dsCache.szEdiUserId);
Index->CacheKey[0].idDataType=EVDT_STRING;
Index->CacheKey[1].nOffset=offsetof(DSCACHE,szEdiBatchNumber);
Index->CacheKey[1].nSize=DIM(dsCache.szEdiBatchNumber);
Index->CacheKey[1].idDataType=EVDT_STRING;
Index->CacheKey[2].nOffset=offsetof(DSCACHE,szEdiTransactNumber);
Index->CacheKey[2].nSize=DIM(dsCache.szEdiTransactNumber);
Index->CacheKey[2].idDataType=EVDT_STRING;
Index->CacheKey[3].nOffset=offsetof(DSCACHE,mnEdiLineNumber);
Index->CacheKey[3].nSize=sizeof(dsCache.mnEdiLineNumber);
Index->CacheKey[3].idDataType=EVDT_MATH_NUMERIC;
Index->CacheKey[4].nOffset=offsetof(DSCACHE.cErrorCode);
Index->CacheKey[4].nSize = 1;
Index->CacheKey[4].idDataType=EVDT_CHAR
```

The flag, idDataType, indicates the data type of the particular key.

This example illustrates a cache with multiple indices and multiple fields:

```
Memset(jdecmIndex,0x00,sizeof(JDECMINDEXSTRUCT)*2);
jdecmIndex[0].nKeyID=1;
jdecmIndex[0].nNumSegments=6;
jdecmIndex[0].CacheKey[0].nOffset=offsetof(I1000042,szCostCenter);
jdecmIndex[0].CacheKey[0].nSize=DIM(dsI1000042.szCostCenter);
jdecmIndex[0].CacheKey[0].idDataType=EVDT_STRING;
jdecmIndex[0].CacheKey[1].nOffset=offsetof(I1000042,szObjectAccount);
jdecmIndex[0].CacheKey[1].nSize=DIM(dsI1000042.szObjectAccount);
jdecmIndex[0].CacheKey[1].idDataType=EVDT_STRING;
jdecmIndex[0].CacheKey[2].nOffset=offsetof(I1000042,szSubsidiary);
jdecmIndex[0].CacheKey[2].nSize=DIM(dsI1000042.szSubsidiary);
jdecmIndex[0].CacheKey[2].idDataType=EVDT_STRING;
jdecmIndex[0].CacheKey[3].nOffset=offsetof(I1000042,szSubledger);
jdecmIndex[0].CacheKey[3].nSize=DIM(dsI1000042.szSubledger);
jdecmIndex[0].CacheKey[3].idDataType=EVDT_STRING;
jdecmIndex[0].CacheKey[4].nOffset=offsetof(I1000042,szSubledgerType);
jdecmIndex[0].CacheKey[4].nSize=1;
jdecmIndex[0].CacheKey[4].idDataType=EVDT_STRING;
jdecmIndex[0].CacheKey[5].nOffset=offsetof(I1000042,szCurrencyCodeFrom);
jdecmIndex[0].CacheKey[5].nSize=DIM(dsI1000042.szCurrencyCodeFrom);
jdecmIndex[0].CacheKey[5].idDataType=EVDT_STRING;
********************** KEY 2 ******************************
jdecmIndex[1].nKeyID=2;
jdecmIndex[1].nNumSegments=7;
jdecmIndex[1].CacheKey[0].nOffset=offsetof(I1000042,szEliminationGroup);
jdecmIndex[1].CacheKey[0].nSize=DIM(dsI1000042.szEliminationGroup);
jdecmIndex[1].CacheKey[0].idDataType=EVDT_STRING;
jdecmIndex[1].CacheKey[1].nOffset=offsetof(I1000042,szCostCenter);
jdecmIndex[1].CacheKey[1].nSize=DIM(dsI1000042.szCostCenter);
jdecmIndex[1].CacheKey[1].idDataType=EVDT_STRING;
jdecmIndex[1].CacheKey[2].nOffset=offsetof(I1000042,szObjectAccout);
jdecmIndex[1].CacheKey[2].nSize=DIM(dsI1000042.szObjectAccount);
jdecmIndex[0].CacheKey[2].idDataType=EVDT_STRING;
jdecmIndex[1].CacheKey[3].nOffset=offsetof(I1000042,szSubsidiary);
jdecmIndex[1].CacheKey[3].nSize=DIM(dsI1000042.szSubsidiary);
jdecmIndex[1].CacheKey[3].idDataType=EVDT_STRING;
jdecmIndex[1].CacheKey[4].nOffset=offsetof(I1000042,szSubledger);
jdecmIndex[1].CacheKey[4].nSize=DIM(dsI1000042.szSubledger);
jdecmIndex[1].CacheKey[4].idDataType=EVDT_STRING;
jdecmIndex[1].CacheKey[5].nOffset=offsetof(I1000042,szSubledgerType);
jdecmIndex[1].CacheKey[5].nSize=1;
jdecmIndex[1].CacheKey[5].idDataType=EVDT_STRING;
jdecmIndex[1].CacheKey[6].nOffset=offsetof(I1000042,szCurrencyCodeFrom);
jdecmIndex[0].CacheKey[6].nSize=DIM(dsI1000042.szCurrencyCodeFrom);
jdecmIndex[0].CacheKey[6].idDataType=EVDT_STRING;
```

2.4.7 Initializing the Cache

After you set up the index or indices, call **jdeCacheInit**, **jdeCacheInitEx**, **jdeCacheInitMultipleIndex**, or **jdeCacheInitMultipleIndexEx**. to initialize (create) the cache. Pass a unique cache name so that JDECACHE can identify the cache. Pass the index to this API so that the JDECACHE knows how to reference the data that will be stored in the cache. Because each cache must be associated with a user, you must also pass the user handle obtained from the call to **JDB_InitUser**. This API returns an HCACHE handle to the cache that JDECACHE creates. This handle appears in every subsequent JDECACHE API to identify the cache.

The keys in the index must be identical for every **jdeCacheInit**, **jdeCacheInitEx**, **jdeCacheInitMultipleIndex**, and **jdeCacheInitMultipleIndexEx** call for that cache until it is terminated. The keys in the index must correspond in number, order, and type for that index each time that it is used.

After the cache has been initialized successfully, JDECACHE operations can take place using the JDECACHE APIs. The cache handle obtained from **jdeCacheInit** or **jdeCacheInitEx** must be passed for every JDECACHE operation. JDECACHE makes an internal Index Definition Structure that accesses the cache when it is populated.

2.4.7.1 Example: Index Definition Structure

In this scenario, assume that each record that the cache stores has this structure:

```
int nInt1
JCHAR cLetter1
JCHAR cLetter2
JCHAR cLetter3
JCHAR szArray(5)
```

The next step is to determine which values to use to index each record in the cache uniquely. In this example, assume that these values are required:

- `nInt1`
- `cLetter1`
- `cLetter3`

Pass that information to **jdeCacheInit** or **jdeCacheInitEx**, and JDECACHE creates this Index Definition Structure for internal use. This table lists Index Definition Structure is for STRUCT letters:

Index Key No.	Index Key Offset	Index Key Offset INTEGER
Index Key #1	0	INTEGER
Index Key #2	4	JCHAR
Index Key #3	6	JCHAR

2.4.8 Using an Index to Access the Cache

When you use an index to access the cache, the keys in the index that are sent to the API must correspond to the keys of the index used in the call to **jdeCacheInit** or **jdeCacheInitEx** for that cache in number, order, offset positions, and type. Therefore, if a field that was used in the index passed to **jdeCacheInit** or **jdeCacheInitEx** offsets position 99, it must also offset position 99 in the index structure that passed to JDECACHE access API.

You should use the same index structure that was used for the call to **jdeCacheInit** or **jdeCacheInitEx** whenever you call an API that requires an index structure.

The next example illustrates why the index offsets must be specified for the **jdeCacheInit** or **jdeCacheInitEx** and how they are used when a record is to be retrieved from the cache. It describes how the passed key is used in conjunction with the JDECACHE internal index definition structure to access cache records.

2.4.8.1 Example: JDECACHE Internal Index Definition Structure

In this example, assume that the user is looking for a record that matches these index key values:

- 1
- c
- i

JDECACHE accesses the values that you pass in the structure at the byte offsets that were defined in the call to **jdeCacheInit** or **jdeCacheInitEx**.

JDECACHE compares the values 1, c, and i that it retrieves from the passed structure to the corresponding values in each of the cache records at the corresponding byte offset. The cache records are stored as the structures that were inserted into the cache by **jdeCacheAdd**, which is the same structure as the one you pass first. The structure that matches the passed key is the second structure to which HCUR1 points.

You should never create a smaller structure that contains just the key to access the cache. Unlike most indexing systems, JDECACHE does not store a cache record's index separately from the actual cache record. This is because JDECACHE deals with memory-resident data and is designed to be as memory-conservative as possible. Therefore, JDECACHE does not waste memory by storing an extra structure for the sole purpose of indexing. Instead, a JDECACHE record has a dual purpose of index storage and data storage. This means that, when you retrieve a record from JDECACHE using a key, the key should be contained in a structure that is of the same type as the structure that is used to store the record in the cache.

Do not use any key structure to access the cache other than the one for which offsets that were defined in the index passed to **jdeCacheInit** or **jdeCacheInitEx**. The structure that contains the keys when accessing a cache should be the same structure that is used to store the cache records.

If **jdeCacheInit** or **jdeCacheInitEx** is called twice with the same cache name and the same user handle without an intermediate call to **jdeCacheTerminate**, the cache that was initialized using the first **jdeCacheInit** or **jdeCacheInitEx** will be retained. Always call **jdeCacheInit** or **jdeCacheInitEx** with the same index each time that you call it with the same cache name. If you call **jdeCacheInit** or **jdeCacheInitEx** for the same cache with a different index, none of the JDECACHE APIs will work.

The key for searches must always use the same structure type that stores cache records.

2.4.9 Using the jdeCacheInit/jdeCacheTerminate Rule

For every **jdeCacheInit**, **jdeCacheInitEx**, **jdeCacheInitMultipleIndex**, or **jdeCacheInitMultipleIndexEx**, a corresponding **jdeCacheTerminate** must exist, except instances in which the same cache is used across business functions or forms. In this case, all unterminated **jdeCacheInit**, **jdeCacheInitEx**, **jdeCacheInitMultipleIndex**, or **jdeCacheInitMultipleIndexEx** calls must be terminated with a **jdeCacheTerminateAll**.

A **jdeCacheTerminate** call terminates the most recent corresponding **jdeCacheInit** or **jdeCacheInitEx**. This means that the same cache can be used in nested business functions. In each function, perform a **jdeCacheInit** or **jdeCacheInitEx** or **jdeCacheInitEx** that passes the cache name. Before exiting that function, call **jdeCacheTerminate**. This does not destroy the cache. Instead, it destroys the association between the cache and the passed HCACHE handle. The cache is completely destroyed from memory only when the number of **jdeCacheTerminate**

calls matches the number of **jdeCacheInit** or **jdeCacheInitEx** calls. In contrast, one call to **jdeCacheTerminateAll** destroys the cache from memory regardless of the number of **jdeCacheInit**, **jdeCacheInitEx**, **jdeCacheInitMultipleIndex**, or **jdeCacheInitMultipleIndexEx** calls or **jdeCacheTerminate** calls.

2.4.10 Using the Same Cache in Multiple Business Functions or Forms

If the same cache is required for two or more business functions or forms, call **jdeCacheInit** or **jdeCacheInitEx** in the first business function or form, and add data to it. After exiting that business function or form, do not call **jdeCacheTerminate** because this removes the cache from memory. Instead, in the subsequent business functions or forms, call **jdeCacheInit** or **jdeCacheInitEx** again with the same index and cache name as in the initial call to **jdeCacheInit** or **jdeCacheInitEx**. Because the cache was not terminated the first time, JDECACHE looks for a cache with the same name and assigns that to you. Because the cache already has records in it, you do not need to refresh it. You can proceed with normal cache operations on that cache.

If a cache is initialized multiple times across business functions or forms, use **jdeCacheTerminateAll** to terminate all instances of the cache that were initialized. The name of the cache that corresponds to the HCACHE passed to this API will be used to determine the cache to destroy. Use this API when you do not want to call **jdeCacheTerminate** for the number of times that **jdeCacheInit** or **jdeCacheInitEx** was called. If you move from one form or business function to another when you initialize the same cache across business functions or forms, you will lose the HCACHE because it is a local variable. To share the same cache across business functions or forms, do not call **jdeCacheTerminate** when you exit a form or business function if you intend to use the same cache in another form or business function.

2.5 Working with JDECACHE Cursors

JDECACHE Cursors (JDECACHE Cursor Manager) is a component of JDECACHE that implements a JDECACHE cursor for record retrieval and update. A JDECACHE cursor is a pointer to a record in a user's cache. The record after the record in which the cursor is currently pointing is the next record that will be retrieved from the cache upon calling a cache fetch API.

This section discusses how to:

- Open a JDECACHE cursor.
- Use the JDECACHE data set.
- Update records.
- Delete records.
- Use the **jdeCacheFetchPosition** API.
- Use the **jdeCacheFetchPostionByRef** API.
- Reset the cursor.
- Close the cursor.
- Use JDECACHE multiple cursor support.
- Use JDECACHE partial keys.

2.5.1 Opening a JDECACHE Cursor

Manipulating the JDECACHE data is cursor-dependent. Before the JDECACHE data manipulation APIs will work, a cursor must be opened. A cursor must be opened to obtain a cursor handle of the type HJDECURSOR, which must, in turn, be passed to all of the JDECACHE data manipulation APIs (with the exception of the **jdeCacheAdd** API). HJDECURSOR is the data type for the cursor handle. It must be passed to every API for JDECACHE data manipulation except **jdeCacheAdd**.

To open the cursor, call the **jdeCacheOpenCursor** API. A call to this API also makes possible the calls to all the data manipulation APIs (except for **jdeCacheAdd**). If you do not open the cursor, these APIs will *not* work. With this call, the cursor opens a JDECACHE data set, within which it will work. This API opens the data set, but does not fetch any data. This means that the cache must be initialized by a call to **jdeCacheInit** or **jdeCacheInitEx** and populated by a call to **jdeCacheAdd** before a cursor can be opened.

You can obtain multiple cursors to a cache by calling **jdeCacheOpenCursor** and passing different HJDECURSOR handles. In a multiple cursor environment, all the cursors are independent of each other.

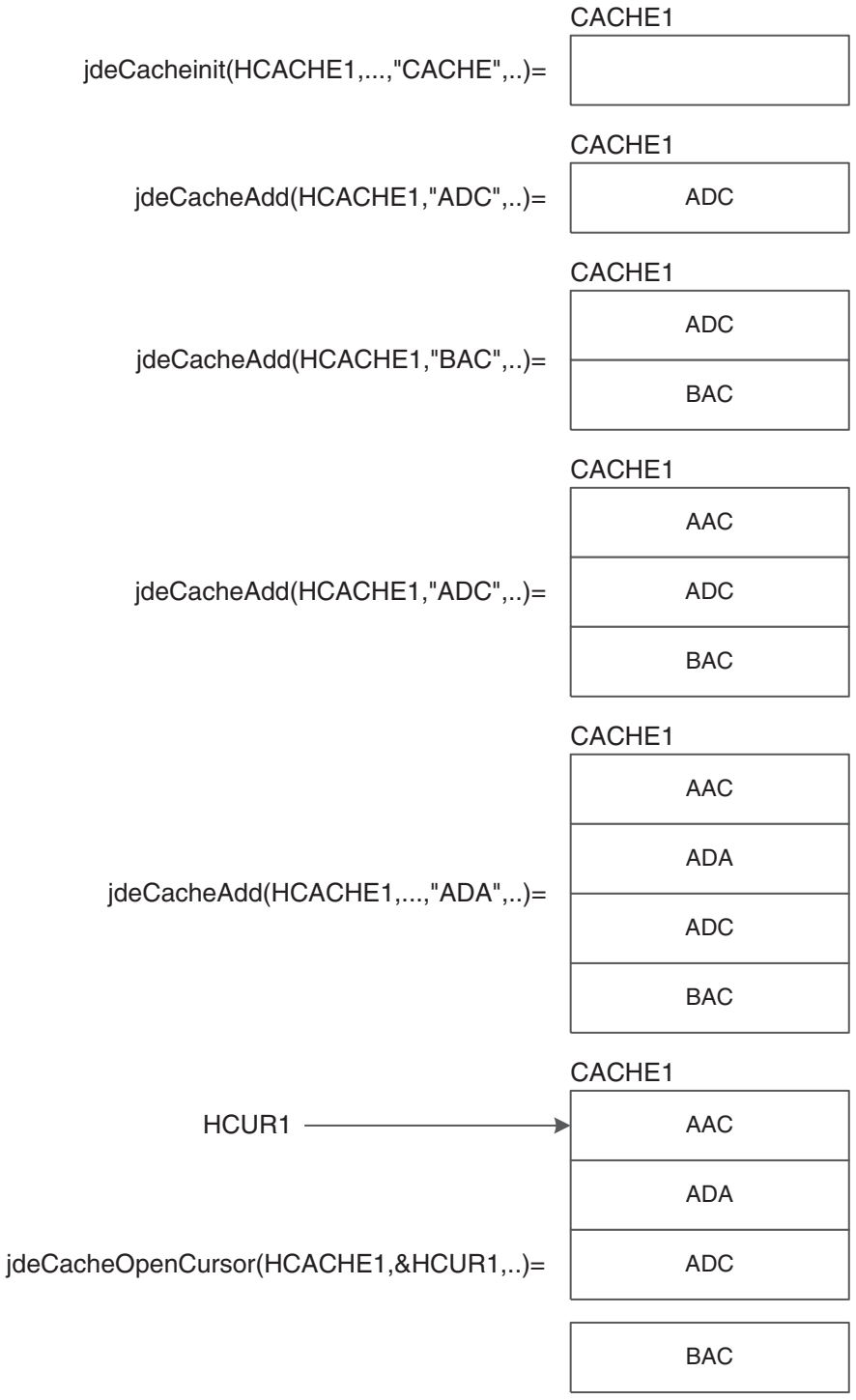
When you are finished working with the cursor, you must deactivate it or close it by calling the **jdeCacheCloseCursor** API, and passing an HJDECURSOR handle that corresponds to the HJDECURSOR handle that was passed to the **jdeCacheOpenCursor**. When a cursor is closed, it cannot be used again until it is opened by a call to **jdeCacheOpenCursor**.

2.5.2 Using the JDECACHE Data Set

The JDECACHE data set includes all of the records from the current position of the cursor to the end of the set of sequenced records. Thus, if a cursor is in the middle of the data set, none of the records in the cache prior to the current position of the cursor is considered part of the data set. The JDECACHE data set consists of the cache records sequenced in ascending order of the given index keys. This means that the order in which the records have been placed in JDECACHE is not necessarily the order in which JDECACHE Cursors retrieves them. JDECACHE Cursors retrieves records in a sequential ascending order of the index keys. A forward movement by the cursor reduces the size of the data set during sequential retrievals. When the cursor advances past the last record in the data set, a failure is returned.

This example illustrates the creation of a JDECACHE cache and a JDECACHE data set:

Figure 2–1 Example of JDECACHE cache and data set creation



2.5.2.1 Cursor-Advancing APIs

Cursor-advancing JDECACHE fetch APIs implement the fundamental concepts of a cursor. The cursor-advancing API set consists of APIs that advance the cursor to the next record in the JDECACHE data set before fetching a record from JDECACHE. **jdeCacheFetch** and **jdeCacheFetchPosition** are examples of cursor-advancing fetch APIs.

A call to **jdeCacheFetch** first positions the cursor at the next record in the JDECACHE data set before retrieving it. JDECACHE Cursors also enable calls to position the cursor at a specific record within the data set. To do this, you call the **jdeCacheFetchPosition** API, which advances the cursor to the record that matches the given key before retrieving it.

You can use a combination of cursor-advancing fetch APIs if you need a sequential fetch of records starting from a certain position. Call **jdeCacheFetchPosition**, passing the key of the record from which you want to start retrieving. This advances the cursor to the desired location in the data set and retrieves the record. All subsequent calls to **jdeCacheFetch** will fetch records starting from the current cursor position in the data set until the end of the data set, or until the program stops for another reason.

2.5.2.2 Non-Cursor-Advancing APIs

Non-cursor-advancing JDECACHE cursor APIs do not advance the cursor before retrieving a record. Instead, they keep the cursor pointing to the retrieved record. **jdeCacheUpdate** and **jdeCacheDelete** are examples of non-cursor-advancing fetch APIs.

2.5.3 Updating Records

If you want to update a specific record with a key that you know, call **jdeCacheFetchPosition**, passing the known key, to position the cursor at the location of the record that matches the key. Because the cursor is already pointing to the desired location, call **jdeCacheUpdate**, passing the same HJDECURSOR that you used in the call to **jdeCacheFetchPosition**.

If the index key changes, cache resorts the records, and the cursor points to the updated location. However, when you call **jdeCacheFetch**, the system retrieves the next record in the updated set. Consequently, the system might not retrieve the correct record because the changed index key caused the order of the records to change.

To update a sequential number of records, make a call to **jdeCacheFetchPosition** to return to the beginning of the sequence, if necessary. Then call **jdeCacheUpdate**, passing the same HJDECURSOR that you used in the call to **jdeCacheFetchPosition**. This call updates only the record to which the cursor is pointing. To update the rest of the records in the sequence, call **jdeCacheFetch** repeatedly, passing the same HJDECURSOR that you used in the call to **jdeCacheFetchPosition**, until you get to the end of the sequence. A sequential update will not work correctly if you have changed any index key value. However, a sequential update will work correctly if you are updating a value that is not an index key.

2.5.4 Deleting Records

If you want to delete a specific record with a known key, first call **jdeCacheFetchPosition** to point the cursor to the location of the record that matches the key. Next, call **jdeCacheDelete**, to remove the record from cache. Pass **jdeCacheDelete** the same HJDECURSOR that you used when you called **jdeCacheFetchPosition**. After deleting a record, use **jdeCacheFetch** to retrieve the record that followed the now-deleted record. This process works only when you call **jdeCacheDelete**.

You can also delete a specific record by calling **jdeCacheDeleteAll** and passing it the full key with the specific record to be deleted. In this case, **jdeCacheFetch** will not work following **jdeCacheDeleteAll**, although you can work around this condition with **jdeCacheFetchPosition** or **jdeCacheResetCursor**.

To delete a sequential set of records, first call **jdeCacheFetchPosition** to point the cursor to the first record in the set or call **jdeCacheDeleteAll** to delete the first record in the set. Then, call **jdeCacheDelete** sequentially. In this case, **jdeCacheFetch** will not work following **jdeCacheDeleteAll**, although you can work around this condition with **jdeCacheFetchPosition** or **jdeCacheResetCursor**.

If you want to delete records that match a partial key, call **jdeCacheDeleteAll** and pass it a partial key. The system deletes all of the records that match the partial key. After you call this API, **jdeCacheFetch** does not work.

2.5.5 Using the jdeCacheFetchPosition API

The **jdeCacheFetchPosition** API searches for a specific record in the data set; therefore, it requires a specific key. This API can perform full and partial key searches.

Note: If you pass 0 for the number of keys, the system assumes that you want to perform a full key search.

2.5.6 Using the jdeCacheFetchPositionByRef API

The **jdeCacheFetchPositionByRef** API returns the address of a data set. The API finds the one record in cache and returns a reference (pointer) to the data. **jdeCacheFetchPositionByRef** retrieves a single, large block of data that is stored in cache. If the cache is empty or has more than one record, this API fails.

2.5.7 Resetting the Cursor

JDECACHE cursors supports multiple cursors, as well as an unlimited number of cursor oscillations within the data set. This means that the cursor can shuttle from beginning to end for an unlimited number of times. The cursor moves forward only. To reset the cursor (move the cursor back to the beginning of the data set), you must make a call to the **jdeCacheResetCursor** API to get a fresh JDECACHE data set.

You can also reset a cursor to a specific position that is outside of the current data set by calling the **jdeCacheFetchPosition** API.

2.5.8 Closing the Cursor

When you no longer need the cursor, call **jdeCacheCloseCursor** to close it. This call closes both the data set and the cursor. Any subsequent call to any JDECACHE API passing the closed HJDECURSOR without having called **jdeCacheOpenCursor** will fail.

Although opening a JDECACHE Cursor for a long period of time requires no overhead, to release the memory that it requires, you should close the cursor as soon as you no longer need it.

2.5.9 Using JDECACHE Multiple Cursor Support

JDECACHE supports multiple open cursors. Each cache that you initialize with **jdeCacheInit** or **jdeCacheInitMultipleIndex** enables up to 100 open cursors to access it at the same time. When you initialize a cache with **jdeCacheInitEx** or **jdeCacheInitMultipleIndexEx**, you can enable any number of cursors, between one and 100, to access it at the same time.

JDECACHE multiple cursors are designed to enable two or more asynchronously processing business functions to use one cache. Asynchronously processing business

functions can open cursors to access the cache with relative positions within the cache that are independent of each other. A cursor movement by one business function does not affect any other open cursor.

Some JD Edwards EnterpriseOne software applications groups restrict the use of multiple cursors. For example, use multiple cursors only if you have a need for them. Additionally, do not use two cursors to point to the same record at the same time unless both cursors are fetching the record.

2.5.10 Using JDECACHE Partial Keys

A JDECACHE partial key is a subset of a JDECACHE key that is ordered in the same way as the defined index, beginning with the first key in the defined index. For example, for a defined index of N keys, the partial key is the subset of the keys 1, 2, 3, 4...N-1 in that specific order. The order is critical. Partial key components must appear in the same order as the key components in the index. (The index is passed to **jdeCacheInit** or **jdeCacheInitEx**.)

For example, suppose that an index is defined as a structure containing the fields in this order: A, B, C, D, E. The partial keys that can be synthesized from this index are this, in order: A, AB, ABC, ABCD. The previous set is the only set of partial keys that can be synthesized for the defined index: A, B, C, D, E.

A JDECACHE partial key implements the JDECACHE cursor. When you implement the JDECACHE partial key, consider that the JDECACHE cursor works within a JDECACHE data set, which comprises the records within the cache *ordered by* the defined index, *the full index*. If you call a jdeCacheFetchPosition API and pass the partial key, the JDECACHE cursor activates and points to the first record in the JDECACHE data set that matches the partial key. If a **jdeCacheFetchPosition** API was called, subsequent calls to **jdeCacheFetch** will fetch all of the records in the data set that succeed the fetched record *to the end of the data set*. The cursor does *not* stop on the last record that matches the partial key, but continues on to fetch the next record using the next call to **jdeCacheFetch**, even if it does not match the partial key. When a partial key is sent to **jdeCacheFetchPosition**, it merely indicates from where the JDECACHE begins fetching. Because the records in the JDECACHE data set are always ordered, the fetch always retrieves all of the records that satisfy the partial key first.

JDECACHE knows that you are passing a partial key because the fourth parameter to **jdeCacheFetchPosition** indicates the number of key fields that are in the key being sent to the API. If the number of key fields is less than the keys that were indicated when **jdeCacheInit** or **jdeCacheInitEx** was called, then it is a partial key. Suppose the number of keys is N so that JDECACHE uses the first N key fields to make comparisons in order to achieve the partial key functionality. If **jdeCacheFetchPosition** is called with a number of keys that is greater than the number specified on the call to **jdeCacheInit** or **jdeCacheInitEx**, an error is returned.

To delete a partial key, you must make a call to **jdeCacheDeleteAll**. This call deletes all of the records that match the partial key. To indicate to JDECACHE the partial keys that you are using, pass the number of key fields to this API.

Verify that the actual number of key fields in the structure corresponds to the numeric value that describes the number of keys that must be sent to either **jdeCacheFetchPosition** or **jdeCacheDeleteAll**.

3

Using Business Functions

This chapter contains the following topics:

3.1 Understanding Business Functions

You can use business functions to enhance JD Edwards EnterpriseOne applications by grouping related business logic. Journal Entry Transactions, Calculating Depreciation, and Sales Order Transactions are examples of business functions.

You can create business functions using one of these methods:

- Event rules scripting language.

  The business functions that you create using the event rules scripting language are referred to as Business Function Event Rules (also called Named Event Rules (NERs)). If possible, use NERs for the business functions. In some instances, C business functions might better suit your needs.

- C programming code.

  JD Edwards EnterpriseOne software creates a shell into which you insert logic using C. You use C business functions mainly for caching, but they can also be used for these objects:

  – Batch error level messaging.

  – Large functions.

    C business functions work better for large functions (as determined by the group). If you have a large function, you can break the code up into smaller individual functions and call them from the larger function.

  – Functions for which performance is critical.

  – Complex select statements.

After you create business functions, you can attach them to JD Edwards EnterpriseOne applications to provide additional power, flexibility, and control. You can attach tables and functions to a business function. You must add related tables and functions to the business function object to generate the code for the source and header files. Because the source code for NERs is generated into C, you use the same procedures for debugging both C and NERs.

This section discusses:

- The components of a business function.
- How distributed business functions work.
- C business functions.
- Business function event rules.

3.1.1 Components of a Business Function

The process of creating a business function produces several components. The Object Management Workbench (OMW) is the entry point for the tools that create the components. These components are created:

Component	Where Created
Business Function Specifications	OMW Business Function Design
Data Structure Specifications	OMW Data Structure Design Tool
.C file	Generated in Business Function Design Modified with the IDE
.H file	Generated in Business Function Design Modified with the IDE

The DLLs are divided into categories. This distribution provides better separation between the major functional groups, such as tools, financials, manufacturing, distribution, and so on. Most business functions are organized into a consolidated DLL based on their system code. For example, a financials business function with system code 01 belongs in CFIN.DLL.

Follow these guidelines when you add or modify business functions:

- Create a custom parent DLL unless you are adding a JD Edwards EnterpriseOne business function.

  Assign a parent DLL to the business functions based on the system code defined in UDC table H92/PL. If no DLL is assigned for the system code in which the business function is created, use CCUSTOM, where CUSTOM is the 7-character version of the company name. You can change the DLL after the business function is created.

- When you write business function code, ensure that all calls to other business functions use the jdeCallObject protocol.

  Linker errors might occur if you do not use **jdeCallObject** and you attempt to call a business function in a different DLL. A linker error prevents the function call from working.

Note: If you change the DLL for a business function, go to C:\B9\System\Bin32\BusBuild.exe, select the old DLL file where the business function was, and select Build from the Build menu to rebuild the file.

This table lists some of the DLLs for which Business Function Builder manages the builds:

DLL Name	Functional Group
CAEC	Architecture
CALLBSFN	Consolidate BSFN Library
CBUSPART	Business Partner
CCONVERT	Conversion Business Functions
CCORE	Core Business Functions
CCRIN	Cross Industry Application
CDBASE	Tools - Database
CDDICT	Tools - Data Dictionary
CDESIGN	Design Business Functions
CDIST	Distribution
CFIN	Financials
CHRM	Human Resources
CINSTALL	Tools Install
CINV	Inventory
CLOC	Localization
CLOG	Logistics Functions
CMFG	Manufacturing
CMFG1	Manufacturing - Modification BFs
CMFGBASE	Manufacturing Base Functions
COBJLIB	Tools - Object Librarian
COBLIB	Busbuild Functions
COPBASE	Distribution/Logistic Base Functions
CRES	Resource Scheduling
CRUNTIME	Tools - Run Time
CSALES	Sales Order
CTOOL	Tools - Design Tools
CTRAN	Transportation
CTRANS	Tools - Translations
CWARE	Warehouse
CWRKFLOW	Tools - Workflow
JDBTRG1	Table Trigger Library 1

DLL Name	Functional Group
JDBTRG2	Table Trigger Library 2
JDBTRG3	Table Trigger Library 3
JDBTRG4	Table Trigger Library 4
JDBTRIG	Parent DLL for Database Triggers

Note: Do not use table triggers for regular business functions.

3.1.2 How Distributed Business Functions Work

OMW manages these three main components that make up NERs or business functions:

- Object Name

  The Object Name is the actual source file.

- Function Name

  The name of the business function or event rule.

  Note: Any business function, whether it uses C or NERs as its source language, must have a defined data structure to send or receive parameters to or from applications. You can create a DSTR data structure object, or select an existing object type to work with in OMW. You can also create data structures for text substitution messages. Additionally, you can attach notes, such as an explanation of use, to any data structure or data item within the structure.

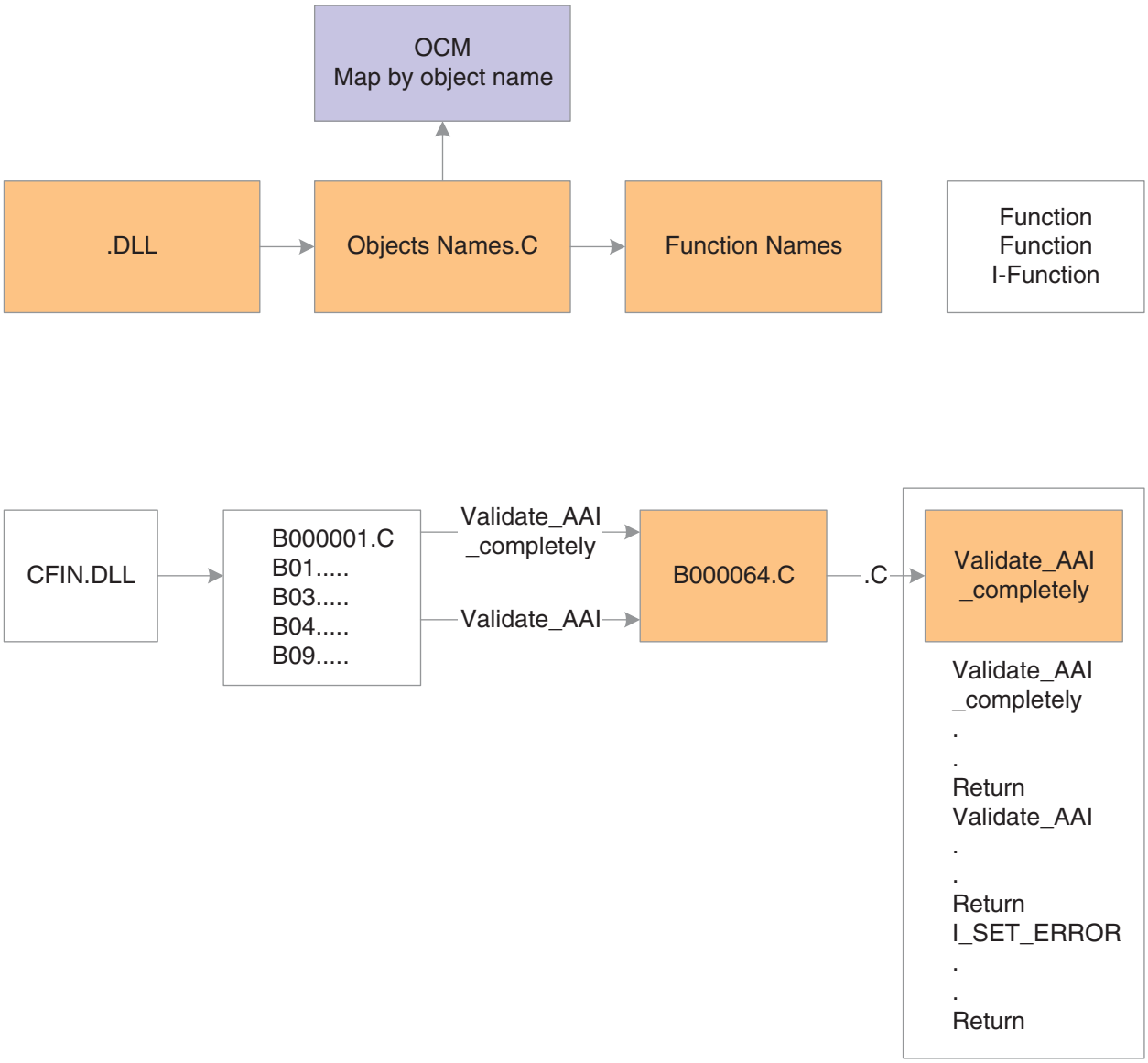
- DLL Name

  The DLL is a dynamic link library.

When a business function is called, the Object Configuration Manager (OCM) determines where to run the business function. After the system maps a business function to a server, calls from that business function cannot be mapped back to the workstation.

This flowchart illustrates how distributed business functions work:

Figure 3–1 Distributed business function



3.1.3 C Business Functions

JD Edwards EnterpriseOne software contains two types of business functions: NERs and C business functions. C business functions are written in C programming language and are used to perform functions that are not available in NERs. C business functions include both a header file (.h) and a source file (.c).

3.1.3.1 Header File Sections

This table describes the major sections of a business function header file:

Section	What It Includes	Description and Guidelines
Header File Comment	■ Header file name ■ Description ■ History ■ Programmer ■ SAR number ■ Copyright information	Comments that the input process of the Business Function Source Librarian builds. The programmer name and SAR number are manually updated by the programmer.
Table Header Inclusions	Include statements for header files associated with tables that are directly accessed by this business function.	Table header files include definitions for the fields in a table and the ID of the table itself.
External Business Function Header Inclusions	Include statements for headers associated with externally defined business functions that are directly accessed by this business function.	External function calls with **jdeCallObject** are included to use the predefined data structures.
Global Definitions	Global constants used by the business function.	Use global definitions sparingly. They include symbolic names that you enter in uppercase; words are separated by an underscore character.
Structure Type Definitions	Data structure definitions for internal processing.	To prevent naming conflicts, define this structure using structure names that are prefixed by the source file name.
DS Template Type Definition	Data structure type definitions generated by Business Function Design. Symbolic constants for the data structure generated by Business Function Design.	Modify this structure through OMW.
Source Preprocessor	■ Undefines JDEBFRTN if it is already defined. ■ Checks for how to define JDEBFRTN. ■ Defines JDEBFRTN.	Ensures that the business function declaration and prototype are properly defined for the environment and source file, including this header.
Business Function Prototype	Prototypes for all business functions in the source file.	Defines the business functions in the source file, the parameters that are passed to them, and the type of value that they return.
Internal Function Prototype	Prototypes for all internal functions that are required to support business functions within this source file.	Defines the internal functions that are associated with the business functions in the source file, the parameters that are passed to each internal function, and the type of value that they return.

3.1.3.2 Example: Business Function Header File

Assume that Business Function Design created this header file. This file contains only the required components in a business function header file:

```
Header File Begin
/*****************************************************************************
* Header File: B99TEST.h
*
* Description: test Header File
*
* History:
*  Date Programmer SAR# - Description
*  ---------- ---------- ----------------------------------------
* Author 10/14/2003 DEMO Unknown - Created
*
*
* Copyright (c) 1994 Oracle 2003
*
* 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 __B99TEST_H
#define __B99TEST_H
/*****************************************************************************
* Table Header Inclusions
*****************************************************************************/
/*****************************************************************************
* External Business Function Header Inclusions
*****************************************************************************/
/*****************************************************************************
* Global Definitions
*****************************************************************************/
/*****************************************************************************
* Structure Definitions
*****************************************************************************/
/*****************************************************************************
* DS Template Type Definitions
*****************************************************************************/
/***************************************
* TYPEDEF for Data Structure
* Template Name: Test Data Structure
* Template ID: D59TEST
* Generated: Tue Oct 14 16:53:08 2003
*
* DO NOT EDIT THE FOLLOWING TYPEDEF
* To make modifications, use the EnterpriseOne Data Structure
* Tool to Generate a revised version, and paste from
* the clipboard.
*
***************************************/
#ifndef DATASTRUCTURE_D59TEST
#define DATASTRUCTURE_D59TEST
typedef struct tagDSD59TEST
{
 JCHAR  cEverestEventPoint01;
 JCHAR  szNameAlpha[41];
 MATH_NUMERIC mnAmountField;
```

```
} DSD59TEST, *LPDSD59TEST;
#define IDERRcEverestEventPoint01_1  1L
#define IDERRszNameAlpha_2   2L
#define IDERRmnAmountField_3   3L
#endif
/*****************************************************************************
* 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 F0101Test
 (LPBHVRCOM lpBhvrCom, LPVOID lpVoid, LPDSD0100018 lpDS);
/*****************************************************************************
* Internal Function Prototypes
*****************************************************************************/
#endif /* __B99TEST_H */
Header File End
```

This table describes the contents of the various lines in the header file:

Header File Line	Where Input	Description
Header File	OMW	Verify the name of the business function header file.
Description	OMW	Verify the description.
History	IDE	Manually update the modification log with the programmer name and the appropriate SAR number.
#ifndef	Business Function Design	Symbolic constant prevents the contents from being included multiple times.
Table Header Inclusion	Business Function Design	When business functions access tables, related tables are input and Business Function Design generates an include statement for the table header file.
External Business Function Header Inclusions	Business Function Design	No external business functions for this application.

Header File Line	Where Input	Description
Global Definitions	IDE	Constants and definitions for the business function. It is not recommended that you use this block. Global variables are not recommended. Global definitions go in .c not .h.
Structure Definitions	IDE	Data structures for passing information between business functions, internal functions, and database APIs.
TYPEDEF for Data Structure	Business Function Design	Data structure type definition. Used to pass information between an application or report and a business function. The programmer places it on the clipboard and pastes it in the header file. Its components include: ■ Comment Block, which describes the data structure. ■ Preprocessor Directives, which ensure that the data type is defined only once. ■ Typedef, which defines the new data type. ■ #define, which contains the ID to be used in processing if the related data structure element is in error. ■ #endif, which ends the definition of the data structure type definition and its related information.
Source Preprocessor Definitions	Business Function Design	All business function header files contain this section to ensure that the business function is prototyped and declared based on where this header is included.
Business Function Prototype	Business Function Design	Used for prototypes of the business function.

Header File Line	Where Input	Description
JDEBFRTN(ID) JDEBFWINAPI CheckForInAddMode	Business Function Design	Business Function Standard All business functions share the same return type and parameter data types. Only the function name and the data structure number vary between business functions. Parameters include: ■ LPBHVRCOM Pointer to a data structure used for communicating with business functions. Values include an environment handle. ■ LPVOID Pointer to a void data structure. Currently used for error processing; will be used for security in the future. ■ LPDS##### Pointer to a data structure containing information that is passed between the business function and the application or report that invoked it. This number is generated through Object Librarian. ■ JDEBFRTN(ID)JDEBFWINAPI All business functions will be declared with this return type. It ensures that they are exported and imported properly. Parameter names (lpBhvrCom, lpVoid, and lpDS) will be the same for all business functions.
Internal Function Prototypes	Business Function Design	Internal function prototypes required to support the business functions in this source file.

3.1.3.3 Source File Sections

OMW builds a template for the business function source file. The business function source file consists of several major sections, as described in this table:

Section	What It Includes	Description
Source File Comment Block	■ Source file name ■ Description ■ History ■ Programmer ■ Date ■ SAR Number ■ Description ■ Copyright information	Built from the information in the Business Function Design Tool. The programmer manually updates the programmer name and SAR number.
Notes Comment Block	Any additional relevant notes concerning the business function source.	Document complex algorithms used, how the business functions in the source relate to each other, and so on.
Business Function Comment Block	■ Business function name ■ Description ■ Description list of the parameters	n/a
Business Function Source Code	Source code for the business function.	n/a
Internal Function Comment Block	■ Function name ■ Notes ■ Returns ■ Parameters	Copy these blocks and place the values in the specified sections to describe the internal function. Follow the comment block with internal function source code.
Internal Function Source Code	Source code for the internal function described in the comment block.	The business function developer enters this code as needed. A populated internal function comment block must precede this code.

3.1.3.4 Example: Business Function Source File

Assume that Business Function Design created this source file called Check for In Add Mode. It contains the minimum components required in a business function source file. The source code in the Main Processing section is entered manually, and varies from business function to business function. All other components are generated by Business Function Design.

```
#include <jde.h>

#define b98sa001_c

/**************************************************************************
 * Source File: B98SA001.c
 *
 * Description: Check for In Add Mode Source File
 **************************************************************************/
 **************************************************************************/

#include <b98sa001.h>

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

```
 * Business Function: CheckForInAddMode
 *
 *  Description: Check for In Add Mode
 *
 *   Parameters:
 *   LPBHVRCOM   lpBhvrCom Business Function Communications
 *   LPVOID    lpVoid  Void Parameter - DO NOT USE!
 *   LPDSD98SA0011   lpDS   Parameter Data Structure Pointer
 *
 ************************************************************************/

JDEBFRTN(ID) JDEBFWINAPI CheckForInAddMode (LPBHVRCOM lpBhvrCom, LPVOID lpVoid,⇒
 LPDSD98SA0011 lpDS)
{
 /************************************************************************
 * Variable declarations
 ************************************************************************/

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

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

 /************************************************************************
 * Check for NULL pointers
 ************************************************************************/
 if ((lpBhvrCom == NULL) ||
  (lpVoid == NULL) ||
  (lpDS  == NULL))
 {
  jdeSetGBRError (lpBhvrCom, lpVoid, (ID) 0, _J("4363"));
  return CONTINUE_GBR;
 }

 /************************************************************************
 * Set pointers
 ************************************************************************/

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

if (lpBhvrCom->iBobMode == BOB_MODE_ADD)
{
 lpDS->cEverestEventPoint01 = _J('1');
}
else

{
 lpDS->cEverestEventPoint01 = _J('0');
}

 return (BHVR_SUCCESS);
}

/* Internal function comment block */
/************************************************************************
```

```
* Function: Ixxxxxxx_a // Replace "xxxxxxx" with source file number
*       // and "a" with the function name
*  Notes:
*
* Returns:
*
* Parameters:
*****************************************************************************/
```

The lines that appear in the source file are described in this table:

Source File Line	Where Input	Description and Guidelines
#include <jde.h>	Business Function Design	Includes all base JD Edwards EnterpriseOne definitions.
#define b98sa001_c	Business Function Design	Ensures that related header file definitions are correctly created for this source file.
Source File	OMW	Verifies the information in the file comment section. Enter the programmer's name, SAR number, and description.
#include <B98SA001.h>	OMW	Includes the header file for this application.
Business Function	Business Function Design	Verifies the name and description in the business function comment block.
JDEBFRTN(ID) JDEBFWINAPI CheckForInAddMode (LPBHVRCOM lpBhvrCom, LPVOID lpVoid, LPDS104438 lpDS)	Business Function Design	Includes the header of a business function declaration.
Variable declarations	IDE	Declares variables that are local to the business function.
Declare structures	IDE	Declares local data structures to communicate between business functions, internal functions, and the database.
Declare pointers	IDE	Declares pointers.
Check for NULL pointers	Business Function Design	Business Function Standard Verifies that all communication structures between an application and the business function are valid.
jdeErrorSet (lpBhvrCom, lpVoid, (ID) 0, _J("4363"), LPVOID) NULL); return ER_ERROR;	Business Function Design	Sets the standard error to be returned to the calling application when any of the communication data structures are invalid.
Set pointers	IDE	Declares and assigns appropriate values to pointers.

Source File Line	Where Input	Description and Guidelines
Main Processing	IDE	Provides main functionality for a business function.
Function Clean Up	IDE	Frees any dynamically allocated memory.
Internal function comment block	IDE	Defines internal functions that are required to support the business function. They should follow the same C coding standards. A comment block is required for each internal function and should be formatted correctly.

Use the MATH_NUMERIC data type exclusively to represent all numeric values in JD Edwards EnterpriseOne software. The values of all numeric fields on a form or batch process are communicated to business functions in the form of pointers to MATH_NUMERIC data structures. MATH_NUMERIC is used as a data dictionary (DD) data type.

3.1.4 Business Function Event Rules

A NER is a business function object for which the source language is event rules instead of C. You create a NER using the event rules scripting language. This scripting language is platform-independent and is stored in a database as a JD Edwards EnterpriseOne software object. NERs are modular. That is, they can be reused in multiple places by multiple programs. This modularity reduces rework and enables you to reuse code.

Not all chunks of code should be packaged in a business function module. For example, when code is so specific that it applies only to a particular program, and it is not reused by any other programs, you should leave it in one place instead of packaging it in a business function. You can attach all the logic on a hidden control (**Button Clicked** event) and use a system function to process the logic as needed.

An example of a NER is N3201030. This business function creates generic text and Work Order detail records (for the F4802 table) for a configured work order. Based on the structure of the sales order in the F3296 table, the configured segments for the item on the passed work order and all lower level segments are included in the generic text.

This example illustrates the function as it appears in Event Rules Design:

```
Named Event Rule Begin
//
// Convert the related sales order number into a math numeric. If that fails
// exit the function
//
String, Convert String to Numeric
If VA evt_cErrorCode is equal to "1"
//
// Validate that the work order item is a configured item.
//
F4102 Get Item Manufacturing Information
If VA evt_cStockingType is not equal to "C"
 And BF cSsuppressErrorMessages is not equal to "1"
BF szErrorMessageID = "3743"
Else
BF szErrorMessageID = " "
```

```
//
// Delete all existing "A" records from F4802 for this work order.
//
VA evt_cWODetailRecordType = "A"
F4802.Delete
F4802.Close
//
// Get the segment delimiter from configurator constants.
//
F3293 Get Configurator Constant Row
If VA evt_cSegmentDelimiter is less than or equal to <Blank>
VA evt_cSegmentDelimiter - /
End If
//
F3296.Open
F3296.Select
If SV File_IO_Status     is equal to CO SUCCESS
F3296.FetchNext
//
// Retrieve the F3296 record of the work order item. and determine its key
// sequence by parsing ATSQ looking for the last occurrence of "1". The substring
// of ATSQ to this point becomes the key for finding the lower level configured
// strings
//
If VA evt_mnCurrentSOLine is equal to BF mnRelatedSalesOrderLineNumber
// Get the corresponding record from F32943. Process the results of that fetch
// through B3200600 to add the parent work order configuration to the work order
// generic text.
F32943.FetchSingle
If SV File_IO_Status   is equal to CO SUCCESS
VA evt_szConfiguredString = concat([VA evt_ConfiguredStringSegment01],
[VA evt_ConfiguredStringSegment02])
Confg String Format Segments Cache
End If
//
// Find the last level in ATSQ that is not "00". Note that the first three
// characters represent the SO Line Number to the left of the decimal.
Example:
// SO Line 13.001 will have the ATSQ characters "013". Each configured item can⇒
 have
// 99 lower-level P-Rule items and a total of ten levels. Therefore every pair
// thereafter is tested.
//
VA evt_mnSequencePosition - 1
While VA evt_mnSequencePosition is less than "23"
And VA evt_szCharacterPair is not equal to "00"
VA evt_mnSequencePosition - [VA evt_mnSequencePosition] + 2
VA evt_szCharacterPair = substr([VA evt_szTempATSQ],[VA evt_mnSequencePostion],2)
End While
VA evt_szParentATSQ = substr([VA evt_szTempATSQ],0,[VA evt_mnSequencePosition])
//
// For each record in F3296 for the related sales order, find those with the same
// key substring of ATSQ. Retrieve the associated record from F32943 if
// available and pass the configured string to N3200600 for addition to the work
// order generic text.
//
F3296.FetchNext
Wile SV File_IO_Status  is equal to CO SUCCESS
VA evt_szChildATSQ = substr([VA evt_szTempATSQ],0,[VA evt_mnSequencePosition]}
If VA evt_szChildATSQ is equal to VA evt_szParentATSQ
```

```
F32943.FetchSingle
If SV File_IO_Status   is equal to CO SUCCESS
VA evt_szCongifuredString = concat([VA evt_ConfiguredStringSegment01],
[VA evt_ConfiguredStringSegment02])
Confg String Format Segments Cache
End If
End If
F3296.FetchNext
End Whil
F32943.Close
//
// Unload segments cache into the work order generic text. B3200600 Mode 6
Confg String Format Segments Cache
//
End If
End If
F3296.Close
//
End If
Else
// The related sales order number is invalid. Return an error.
If BF cSuppressErrorMessages is not equal to "1"
Set NER Error ("0002", BF SzRelatedSalesOrderNumber)
End If
End Ir
Named Event Rule End
```

3.2 Understanding Transaction Master Business Functions

Transaction master business functions provide a common set of functions that contain all of the necessary default values and editing for a transaction table in which records depend on each other. Transaction master business functions contain logic that ensures the integrity of the transaction being inserted, updated, or deleted from the database. Event flow breaks up logic. You use cache APIs to store records that are being processed. You should consider using a transaction master business function in these situations:

- You accept transaction file records from a non-JD Edwards EnterpriseOne source.
- Multiple applications update the same transaction file.

These transaction tables are examples of candidates for transaction master business functions:

- The F0911 table accepts updates across application suites, as well as external sources.
- The F06116 table accepts updates from batch, interactive, and external sources.

A master business function (MBF) can be called from several different applications. Rather than duplicating the processing options for the MBF on each application, you typically create a separate processing option template for these processing options. You can use interactive versions to set up different versions of the MBF processing options. Various calling programs then pass the version name to the version parameter of **BeginDoc**.

From within **BeginDoc**, the business function **AllocatePOVersionData** can be called to retrieve the processing options by version name. The processing options needed by other modules can be written to the header cache and accessed later, rather than calling **AllocatePOVersionData** multiple times.

The cache structure stores all lines of the transaction. Transaction lines are written to the cache after they have been edited. The **EndDoc** module then reads the cache to update the database.

This table describes the components of the header section:

Field Description	Field	Key	Type	Size
Job Number	JOBS	X	Num	N/A
Document Action	ACTN	N/A	Char	1
Processing Options	N/A	N/A	N/A	N/A
Currency Flag	CYCR	N/A	Char	1
Business View Fields	N/A	N/A	N/A	N/A
Work Fields	N/A	N/A	N/A	N/A

This table explains the fields:

Field Description	Purpose
Job Number	A unique system-assigned number assigned when the BeginDoc module starts the job. This distinguishes transactions in the cache for each job on the workstation that is using the cache. Use next number 00/4 for the job number. If you are using a unique cache name (Dxxxxxxxxx[job number]), you do not necessarily need the job number field stored in the cache for a key because you would only be working with one transaction per cache. You can, therefore, use any field as the key to the cache.
Document Action	The action for the document. Values are: ■ A or 1 = Add ■ C or 2 = Change ■ D = Delete
Processing Options	Processing option values were read in using **AllocatePOVersionData**, and are needed in other modules of the MBF.
Currency Flag	A system value that indicates whether currency is on and what method of currency conversion is used (N, Y, or Z).
Business View Fields	The fields required for processing the transaction and writing it to the database. All fields in the record format that are not saved in the header cache will be initialized when the record is added to the database using the APIs.

Field Description	Purpose
Work Fields	Fields that are not part of the business view (BV), but are needed for editing and updating the transaction. For example, Last Line Number is the last line number written to the detail cache. It will be stored at the header level, and retrieved and incremented by the MBF. The incremented line number will be passed to the header cache and stored for the next transaction.

This table describes the components of the detail section:

Field Description	Field	Key	Type	Size
Job Number	JOBS	X	Char	8
Line Number	(Application-specific)	X	Num	N/A
Line Action	ACTN	N/A	Char	1
Business View Fields	N/A	N/A	N/A	N/A
Work Fields	N/A	N/A	N/A	N/A

This table explains the fields:

Field Description	Purpose
Job Number	A unique number assigned when the **BeginDoc** module starts the job. This distinguishes transactions in the cache for each job on the client that is using the cache. If you are using a unique cache name (Dxxxxxxxxx[job number]), you do not necessarily need to store the job number field in the cache for a key because you work with only one transaction per cache. You can, therefore, use line number only as the key to the cache.
Line Number	The number used to uniquely identify lines in the detail cache. This line number can also eventually be assigned to the transaction when it is written to the database. The transaction lines are written to the detail cache only if they are error-free.
Line Action	The action for the transaction line. Values are: ■ A or 1 = Add ■ C or 2 = Change ■ D = Delete
Business View Fields	Fields required for processing the transaction that will be written to the database. All fields in the record format that are not saved in the detail cache will be initialized when the record is added to database using the APIs.

Field Description	Purpose
Work Fields	Fields that are not part of the business view, but are needed for editing and updating the transaction line.

3.3 Building Transaction Master Business Functions

This section provides an overview of building transaction master business functions, and discusses the component used to build such a business function:

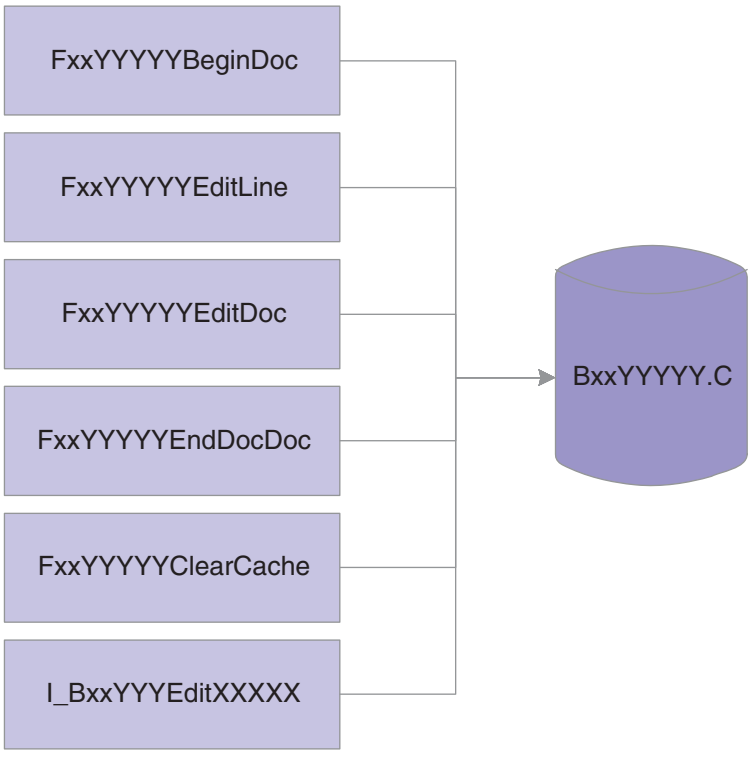
- Begin document
- Edit line
- Edit document
- End document
- Clear cache
- Cancel document

3.3.1 Understanding Building Transaction Master Business Functions

These flowcharts illustrate how transaction master business functions are built.

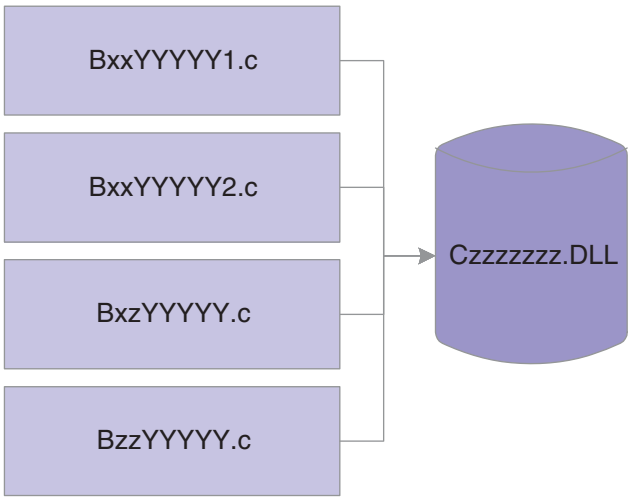
First, you create the individual business functions using several basic components:

Figure 3–2 Building transaction master business functions



Next, you combine the business functions into a DLL:

Figure 3–3 Combining business functions into a .DLL



You typically use these basic components to create a master business function as described by this table:

Component	Purpose
Begin Document	Called when all header information has been entered. Creates initial header if it has not already been created. Can also include default values, editing, and processing options (POs).
Edit Line	Called when all line information has been entered. Creates cache for detail information if it has not already been created.
Edit Document	Called when ready to commit the transaction. Processes any remaining document edits and verifies that all records are valid to commit.
End Document	Called when you need to commit the transaction. Processes all records in the header and detail cache, performs I/O, and deletes caches.
Clear Cache	Called when you are ready to delete all cache records. Deletes header and detail cache records.

3.3.2 Begin Document

Begin Document has this format:

```
FxxxxxBeginDocument
```

The Begin Document component performs these tasks:

- Inserts default information and edits information in the header, including data dictionary defaults and UDC editing.
- Fetches information from the database, if necessary, to ensure that the selected document action can take place.
- Validates and processes information that is common to all records.
- Writes the record to header cache if no errors exist.
- Contains all header cache information that is common to all detail records. This improves performance by eliminating the need to use all the detail records to perform the same validations and table I/O.

- Updates the header cache with the new information when information in the header fields changes and Begin Document has previously been called.

3.3.2.1 Special Logic or Processing Required

On the initial call, the function assigns the job number. To retrieve the job number, this function calls X0010GetNextNumber with a system code of 00 and an index number of 04. If called again, Begin Document passes the job number that was previously assigned; therefore, it does not need to assign another job number.

3.3.2.2 Hook Up Tips

Keep these tips in mind when calling Begin Document:

- You must call a function at least once before calling Edit Line.
- If errors occur during validation of the header field when the function is called, call the function again to verify that errors have been cleared before calling Edit Line.
- If this function might be called multiple times from different events, include it on a hidden button on an application to reduce duplicate code and ensure consistency. This button might then be called from focus on grid because the user is then adding or deleting detail records, and is finished adding header information. In case of a Copy in which the user does not use the grid, this button might also be called on OK button.
- Calling a button from an asynchronous event breaks the asynchronous flow and forces the button to be processed in synchronous mode (inline).

3.3.2.3 Common Parameters

This table describes the common parameters for Begin Document:

Name	Alias	I/O	Description
Job Number	JOBS	I/O	Pass Job Number created in Begin Document, if previously called; otherwise, pass zeros and assign a job number.
Document Action	ACTN	I	A or 1 = Add C or 2 = Change D = Delete This is the action of the entire Document, not the individual detail lines. For example, you might modify a few detail lines in Edit Line, add a few detail lines in Edit Line, and delete a few detail lines in Edit Line, but the Document Action in Begin Document would be Change.

Name	Alias	I/O	Description
Process Edits	EV01	I	Optional 0 = No Edits Any Other = Full Edits **Note:** The GUI interface usually uses the partial edit, and the batch interface uses the full edit. If you leave this parameter blank, the default option is full edits.
ErrorConditions	EV02	O	Blank = No Errors 1 = Warning 2 = Error
Version	VERS	I	This field is required if this MBF is using versions.
Header Field One	****	I/O	Pass in all the header fields that are common to the entire document. Begin Document processes all of these fields and validates them, data dictionary edits, UDC editing, default values, and so on. Begin document might also fetch to the table to validate that records matching these header fields exist for Delete and Change, or do not exist for Add.
Header Field Two	****	I/O	N/A
. . .	****	I/O	N/A
Header Field XX	****	I/O	N/A

Name	Alias	I/O	Description
Work Field / Processing Flag One	****	I	List any work fields that the program needs. These could be flags for processing, dates to validate, and so on. These fields might or might not be used. For example, currency control might be saved in the header cache so that all detail records would either use currency or not.
Work Field / Processing Flag One	****	I	N/A
.	N/A	I	N/A
.			
.			
Work Field / Processing Flag One	N/A	I	N/A

3.3.2.4 Application-Specific Parameters

Application-specific parameters must perform these tasks:

- List the fields that are needed to process header-level information.
- List any work fields that are needed to perform edits.
- List all POs that are needed to process header-level information.

3.3.3 Edit Line

Edit Line has this format:

```
FxxxxxEditLine
```

The Edit Line component performs these tasks:

- Validates all user input, performs calculations, and retrieves default information.

  Edit Line is normally called for every record that is fetched. It performs the edits for that *one* record in the file.

- Reads header cache records for default values.
- On an ADD, enters default information in blank columns, such as address book information.

  The default values might come from any of these objects:

  - Another column in the line.
  - A process performed on a column sent in the line.
  - A PO.
  - A saved value from the header record that was determined in the Begin Document module.
  - A DD default value.

- Edits columns for correct information.

  This includes interdependent editing between columns. Also performs UDC and DD edits.

- Writes record to the detail cache if no errors occurred.

  If the record already exists in the work file, the line in the work file will be retrieved and updated with the changes. If a record is deleted from the grid in direct mode, and the record does not exist in the database, the record will be removed from the detail cache. If the record exists in the database, the action code for the record will be changed to delete, and the record will be stored in the detail cache until file processing in End Doc.

3.3.3.1 Special Logic or Processing Required

Depending on the type of document being processed, different editing and inserting of default values takes place. An example would be vouchers and invoices processed through the journal entry MBF. The tax calculator is only called for vouchers. Depending on the event processing required, the process edit flag determines the editing that occurs. For example, in an interactive program, when the *Grid Record is Fetched* event runs, Partial Edits might be performed to retrieve descriptions, default values, and so on. When the *Row is Exited and Changed* event runs, Full Edits might be performed to validate all user input.

3.3.3.2 Typical Uses and Hookup

In interactive applications, Edit Line is typically called on Grid Record is Fetched or Row is Exited and Change (Asynch). In batch applications, Edit Line is typically called in the Do section of the group, columnar, or tabular section.

3.3.3.3 Common Parameters

This table describes the common parameters for Edit Line:

Name	Alias	I/O	Description
Job Number	JOBS	I	Used as key or to create a unique name for the cache or work file. Retrieved from Begin Document.
Line Number	LNID	I/O	The unique number identifying the transaction line. Can also be used as the line number in the Detail Cache.
Line Action	ACTN	I	A or 1 = Add C or 2 = Change D or 3 = Delete

Name	Alias	I/O	Description
Process Edits (optional)	EV01	I	0 = No Edits 1 = Full Edits 2 = Partial Edits **Note:** GUI interface typically uses the partial edit, and the batch interface typically uses the full edit. If you leave this parameter blank, the default edit is Full.
Error Conditions	ERRC	O	0 = No Errors 1 = Warning 2 = Error
Update Or Write to Work File	EV02	I	1 = Write or update records to the work file, or do both.
Record Written to Work File	EV03	I/O	1 = A record is written to the work file. This reduces I/O calls to the work file. Blank = No record is written to the work file.
Detail Field One	****	I/O	Pass in all the Detail fields that will be edited. Typically, these are the grid record fields. Edit Line provides validation, data dictionary edits, UDC editing, default values, and so on.
Detail Field Two	****	I/O	N/A
Detail Field XX	****	I/O	N/A
Work Field / Processing Flag One	****	I	List any work fields that the program needs. These fields could be flags for processing, dates to validate, and so on.
Work Field / Processing Flag One	****	I	N/A
Work Field / Processing Flag One	****	I	N/A

3.3.4 Edit Document

The Edit Document component performs these tasks:

- Reads cache records if multiple line editing is required.
- Reads header cache record if header information is needed.

- Performs cross-dependency edits involving multiple lines in a document. For example, Edit Document processes all records to ensure that percentages total 100 percent, and it ensures that the last record does not contain certain information.

3.3.4.1 Special Logic or Processing Required

Depending on the type of document that you are processing, different logic is executed. For example, vouchers and invoices are processed through the journal entry edit object, although the balancing is different for these document types.

3.3.4.2 Hook Up Tips

Edit Document is typically used in this fashion:

- Call the function at least once after calling Edit Line and before End Document.
- If errors occur during validation, call the function again to verify that errors have been cleared before calling End Document.
- Call this function on the OK *Button Clicked* event so that, if errors do occur, they are corrected before the user exits the application.

3.3.4.3 Common Parameters

This table describes the common parameters for Edit Document:

Name	Alias	I/O	Description
Job Number	JOBS	I	Retrieved from Begin Document
ErrorConditions	EV01	O	Blank = No Errors 1 = Warning 2 = Error

3.3.4.4 Application-Specific Parameters

Because all records have been added in Begin Document or Edit Line, and because any information needed to process the entire document is in cache, few parameters are needed in this function.

3.3.5 End Document

End Document has this format:

```
FxxxxxEndDocument
```

The End Document component performs these tasks:

- Assigns a next number to the document.

  For vouchers, you should do this before calling journal entry edit object, but not before the voucher has been balanced and is ready to be added to the database. By placing this module on the before add/delete/update events, the document passes all edits before running this event.
- Reads cache records.
- On an ADD, writes new rows to the table.
- On a CHG, retrieves and updates existing rows.
- On a DEL, deletes rows from the table.

- Adds information and updates associated tables.

  For example, it adds and updates these objects:

  - Manual checks associated with vouchers.
  - Address Book vouchered YTD columns in Address Book.
  - Address, phones, and who's who information for Address Book.
  - Batch header.
- Clears the cache for that document and any work fields after all updates are completed successfully.
- Summarizes documents, if designated in a processing option, as it writes to the database.
- Reads work file through an alternate means and writes the records at a control break.
- Performs currency conversion.

3.3.5.1 Hook-Up Tips

This function is typically called on OK button **Post Button Clicked**, and it is hooked up Asynch. In the C code, after the insert or update to the database is successful, call **Clear Cache** to clear the cache.

3.3.5.2 Common Parameters

This table describes the common parameters for End Document:

Name	Alias	I/O	Description
Job Number	JOBS	I	Retrieved from Begin Document
Computer ID	CTID	I	Retrieved from GetAuditInfo(B9800100) in application (optional)
Error Conditions	EV01	O	Blank = No errors 1 = Warning 2 = Error
Program ID	PID	I	Usually hard-coded

3.3.5.3 Application-Specific Parameters

Use application-specific parameters in End Document to perform these tasks:

- List the fields that are needed to process update or writes, such as Time and Date Stamp fields.
- List any work fields that are needed to perform updates or writes.
- List all POs that are needed to process updates or writes.

3.3.6 Clear Cache

Clear Cache has this format:

```
FxxxxxClearCache
```

The Clear Cache component removes the records from the header and detail cache.

3.3.6.1 Special Logic or Processing Required

If a unique cache name is selected as the naming convention for the cache (Dxxxxxxxx[Job Number]), then use the cache API **jdeCacheTerminateAll** to destroy the cache.

3.3.6.2 Common Parameters

This table describes the common parameters for Clear Cache:

Name	Alias	I/O	Description
Job Number	JOBS	I	Indicates the job number of the transaction that you want to clear. This job number should have been returned from BeginDoc.
Clear Header	EV01	I	Indicates whether the header cache should be cleared. 1 = clear cache
Clear Detail	EV02	I	Indicates whether the detail cache should be cleared 1 = clear cache
Line Number From (Optional)	LNID	I	Indicates where to begin clearing records in the detail cache. If this line is blank, the system begins clearing from the first record.
Line Number Thru (Optional)	NLIN	I	Indicates where to stop clearing records in the detail cache. If this line is blank, the system deletes to the end of the cache.

3.3.7 Cancel Document

Cancel Document has this format:

```
FxxxxxxCancelDoc
```

The optional Cancel Document component is used primarily with the Cancel button to close files, clear the cache, and so on. Cancel Document is an application-specific function that provides basic function cleanup.

3.3.7.1 Special Logic or Processing Required

This function is application-specific.

3.3.7.2 Common Parameter

This table describes the common parameter for Cancel Document:

Name	Alias	I/O	Description
Job Number	JOBS	I	The job number of the transaction that you want to clear. This number should have been returned from BeginDoc.

3.4 Implementing Transaction Master Business Functions

This section discusses using single-record processing and document processing to implement transaction master business functions.

3.4.1 Single-Record Processing

This section provides an interactive and a batch program flow example for single-record processing.

3.4.1.1 Interactive Program Flow Example

This is an example of an implementing transaction master business functions during single-record processing in an interactive application:

1. **Post Dialog is Initialized (optional)**

   Call Begin Document.

2. **Set Focus on Grid**

3. **Row is Exited and Changed** or **Row is Exited and Changed ASYNC**

   Call Edit Line.

4. **Delete Grid Record Verify- After**

   Call Edit Line to perform delete for one record.

   Call Edit Document to perform deletes on a group of records.

5. OK **Button Clicked**

   Call Begin Doc.

   Call Edit Document.

6. OK **Post Button Clicked**

   Call End Document.

Master Business Functions usually perform all table I/O for the given table. Therefore, these actions must be disabled:

- **Add Grid Record to DB - before**

  Suppress Add.

- **Update Grid Record to DB - before**

  Suppress Update.

- **Delete Grid Record to DB - before**

Suppress Delete.

3.4.1.2 Batch Program Flow Example

This is an example of an implementing transaction master business functions during single-record processing in a batch application:

1. Do Section of Report Header.

   Call Begin Document.

2. Do Section of the Group Section.

   Call Edit Line.

3. Do Section of a Conditional Section (optional).

   Call Edit Document.

4. Do Section of Report Footer.

   Call End Document.

3.4.2 Document Processing

This section provides an interactive program flow example for document processing.

3.4.2.1 Program Flow Example

This is an example of an implementing transaction master business functions during document processing in an interactive application:

1. **Dialog is Initialized**

   Call Open Batch Edit Object module.

2. **Grid is Entered**

   Call Begin Document Edit Object module.

3. **Row is Exited**

   Call Edit Line Edit Object module.

4. OK **Button Clicked**

   Call Edit Document Edit Object module.

5. **Before Add from Database** or **Before Delete from Database**

   Suppress Add/Delete.

   Call End Document Edit Object module.

6. Cancel **Button Clicked**

   Call Close Batch Edit Object module.

3.5 Working with Master File Master Business Functions

Master business functions (MBFs) enable calling programs to process certain predefined transactions. An MBF encapsulates the required logic, enforces data integrity, and insulates the calling programs from the database structures. Use MBFs for these reasons:

- To create reusable, application-specific code.

- To reduce duplicated code.
- To ensure that hookup is consistent.
- To support interoperability models.
- To enable processing to be distributed through OCM.
- To design event-driven architecture.

MBFs are typically used for multiline business transactions such as journal entries or purchase orders. However, certain master files also require MBF support due to their complexity, importance, or maintenance requirements from external parties. The requirements for maintaining master files are different from those for multiline business transactions.

Generally, master file MBFs are much simpler than multiline business transaction MBFs. Transaction MBFs are specific to a program, while master file MBFs access a table multiple times.

For interoperability, master file MBFs can be used instead of table I/O. This enables you to perform updates to related tables using the business function instead of table event rules. Multiple records are not used; instead, all edits and actions are performed with one call.

In their basic form, master file MBFs have these characteristics:

Characteristic	Description
Single call	Generally, you can make one call to an MBF to edit, add, update, or delete a master file record. An edit-only option is available also.
Single data structure	The fields required to make the request and provide all the necessary values are in one data structure. The data fields should correspond directly with columns in the associated master file.
No cache	Because each master file record is independent of the others, caching is unnecessary. The information provided with each call and the current condition of the database provides all of the information that the MBF needs to perform the requested function.
Normal error handling	As with other MBFs, master file MBFs must be capable of executing both in interactive and batch environments. Therefore, the calling program must determine the delivery mechanism of the errors.
Inquiry feature	To enable external systems to be insulated from the JD Edwards EnterpriseOne database, an inquiry option is included. This enables an external system to use the same interface to access descriptive information about a master file key as it uses to maintain it.
Effect on applications	For JD Edwards EnterpriseOne applications, the effect of implementing a master file MBF should be minimal. Consider and follow several standards before implementing a master file MBF.

Master file applications use the system to process all I/O for find/browse forms. This enables you to use all of the search capabilities of the software.

You should design all master file applications so that all fix/inspect forms are independent of each other. Each fix/inspect form can use the system to fetch the record, and all edits and updates occur using the master file MBF. This independent design has these major benefits:

- It organizes the application in a way that simplifies edits involving dependent fields across multiple forms.
- It enables consistent implementation of modeless processing for all master file applications and all forms within these applications.

Certain circumstances might justify deviation from this simple model. These circumstances are:

- Extremely large file formats

  When the number of columns in the master file plus the required control fields in the call data structure exceed technical limitations for data structures, the MBF can be split. You can split the MBF into one MBF that handles base data and performs all adds and deletes, and one or more MBFs that enable the calling program to update additional data when the base data has been established. In this case, it is usually logical to split it, regardless of the technical limitation. For example, assuming that the customer master file exceeded the data structure limitation, you would use these two MBFs to process the file:

- F0301ProcessMasterData
- F0301ProcessBillingData

  In this example, the F0301ProcessMasterData function processes the base data, and the F0301ProcessBillingData function updates additional data.

- Subordinate detail files

  Information can exist in addition to the primary master file that has been normalized to enable for a one-to-many relationship. Designing the Master File MBF strictly on the basis of how the database is designed translates into three calls. Including at least one occurrence of a detail relationship in the data structure of a Master File MBF is valid. This inclusion enables users to establish reasonably complete master file information using a simple interface to meet simple needs. Street addresses and phone numbers within Address Book are a good example. Customers expect that they can create an address book record by calling a simple address book API with basic identifying information, the street address, and a phone number.

3.5.1 MBF Information Structure

This section discusses the parameters of the MBF information structure.

3.5.1.1 Standard Parameters for Single-Record Master Business Functions

This table describes the standard parameters for single-record MBFs:

Name	Alias	I/O	Required/Optional	Description
Action Code	ACTN	I	Required	A = Add. I = Inquiry. C = Change. D = Delete. S = Same as except (the record is the same except for what the user changes).
Update Master File	EV01	I	Optional	0 = No update; edit only (default). 1 = Update performed.

Name	Alias	I/O	Required/Optional	Description
Process Edits	EV02	I	Optional	1 = All Edits (default). 2 = Partial Edits (no data dictionary (DD)).
Suppress Error Messages	SUPPS	I	Optional	1 = Error messages are suppressed. 0 = Process errors normally (default).
Error Message ID	DTAI	O	Optional	Returns error code.
Version	VERS	I	Future	The default value is XJDE0001.

3.5.1.2 Application-Specific Control Parameters (Example: Address Book)

This table describes the application-specific parameters for Address Book:

Name	Alias	I/O	Required/ Optional	Description
Address Book Number	AN8	I/O	Optional	For additions, AN8 is optional. For all other action codes, this parameter is required.
Same as except	AN8	I	Optional	Required for S = Action Code. The record is the same except for what the user changes.

3.5.1.3 Application Parameters (Example: Address Book)

This table describes the application parameters for Address Book:

Name	Alias	I/O	Required/Optional
Alpha Name	ALPH	I/O	Required
Long Address Number	ALKY	I/O	Optional
Search Type	AT1	I	Required
Mailing Name	MLMN	I	Required
Address Line 1	ADD1	I	Optional
City	CTY1	I	Optional
State	ADDS	I	Optional
Postal Code	ADDZ	I	Optional

3.5.2 Master Business Function Impact on Performance

Performance issues might occur regardless of how you handle large-format tables. Two options for improving performance are:

- Group data logically to enable data structures to be smaller and easier for the user to implement.

  This configuration does, however, force the user to make multiple calls to add or update an entire record in a table.

- Use a data structure that enables 300 fields.

  This configuration is cumbersome to implement, and the user can choose not to apply all of the fields.

Through different interfaces, the user can add additional data later. Most processes dictate that part of the data be added immediately, while related data can be added later. For example, the user might define a customer master record but wait until a later date to define the customer's billing instructions. Therefore, you should select the first option of splitting MBFs so that one MBF handles base data and one MBF handles additional data.

3.6 Working with Business Functions

Every business function must follow a defined structure and form. Every line of code must conform to the JD Edwards EnterpriseOne business function programming standards. Creating a business function involves these overall tasks:

- Use JD Edwards EnterpriseOne Object Management Workbench (OMW) to build business function data structures.
- Use OMW to build business function source and header files.
- Build and add type definitions for data structures to the header file.

Business function DLLs are consolidated. Therefore, you need to build each of the custom business functions into a custom DLL that you create. This process ensures that the custom business functions remain separate from JD Edwards EnterpriseOne business functions. The build program reviews the F9860 table to verify that the custom DLL exists.

When you create a custom business function, you need to specify one of the custom DLLs. If you do not, the build process builds the custom business function into the JD Edwards EnterpriseOne CCUSTOM.DLL, where CCUSTOM is the seven-character name of the company, which is the default.

3.6.1 Prerequisite

Create a data structure.

3.6.2 Creating a Custom DLL

To create a custom DLL:

1. In OMW, create a new Business Function Library.
2. In Windows, run BusBuild.exe.

   Typically, this file is located in ..\B9\System\Bin32\.
3. Rebuild all libraries by selecting Build, Rebuild Libraries in OMW.

   This process takes several minutes.

3.6.3 Specifying a Custom DLL for a Custom Business Function

To specify a custom DLL for a custom business function :

1. In Business Function Design Aid, enter the custom DLL name in the Parent DLL field.

Note: You can also change the business function location if necessary.

---

2. Run the build for the business function.

3.7 Working with Business Function Builder

Use JD Edwards EnterpriseOne Business Function Builder to build business function code into a DLL. You can build C business functions, Named Event Rules (NERs), and table event rules. The process that occurs when you run JD Edwards EnterpriseOne Business Function Builder to build business functions includes compiling and linking. Compiling involves creating a business function object. Linking makes the object part of a DLL.

Note: Link All does not compile any business functions; it only links each DLL.

You usually use JD Edwards EnterpriseOne Business Function Builder to build a single business function. Whenever you create source code changes to a business function, you must build the business function to test it.

Build Output displays the results of the build. When the build is finished, the message `***Build Finished***` appears at the bottom of Build Output. The text after this line indicates whether the build was successful. If the build was successful, you can test the business function. Otherwise, you must correct any problems and rerun the build process.

The system creates a work directory when any object is built. This directory is in the destination directory that you specified, such as C:\b7\appl_pgf\work\buildlog.txt. This directory contains error and information logs. The build log contains the same information as the Build Output form in JD Edwards EnterpriseOne Business Function Builder.

3.7.1 Setting Build Options

Use options on the Build menu to control how and when the consolidated business function is built. This table describes the available options:

Option	Result
Build	Generates a makefile, compiles the selected business functions, and links the functions into the current consolidated DLL. Rebuilds only those components that are out of date.
Compile	Generates a makefile and compiles the selected business functions. The application does not link the functions into the current consolidated DLL.
ANSI Check	Reviews the selected business function for ANSI compatibility.
Link	Generates a makefile for each consolidated DLL and then builds each consolidated DLL. The application does not compile any of the selected business functions.
Link All	Generates a makefile for each consolidated DLL and then builds each consolidated DLL and links it to all business functions that are called. The application does not compile any of the selected business functions.
Rebuild Libraries	Rebuilds the consolidated DLL and static libraries from the .obj files.
Build All	Links and compiles all objects within each DLL.
Stop Build	Stops the build from finishing. The existing consolidated DLL remains intact.

Option	Result
Suppress Output	Limits the text that appears in Build Output.
Browse Info	Generates browse information when compiling business functions. Clear this option to expedite the build.
Precompiled Header	Creates a precompiled header when compiling a business function. When compiling multiple business functions, the Business Function Builder generally compiles faster if it uses a precompiled header.
Debug Info	Generates debug information when compiling. The Visual C++ can debug any function that was built with debug information. Clear this option to expedite the build.
Full Bind	Resolves all of the external runtime references for each JD Edwards EnterpriseOne consolidated DLL.

3.7.2 Reading Build Output

Build Output consists of a series of sections that display important information about the status of a build. You can use this information to determine whether the build completed successfully and to troubleshoot problems if errors occurred during the build.

3.7.2.1 Makefile Section

The makefile section indicates where Business Function Builder generated the makefile for a particular build. JD Edwards EnterpriseOne Business Function Builder generates one makefile for each DLL that it builds. A *Generating Makefile* statement should always appears for each DLL that you are building. If the makefile statement does not appear, then an error occurred. To resolve the error, you must complete these tasks:

- Verify that the local object directory exists.
- Verify that the permissions for the local object directory and the makefile are correct.

3.7.2.2 Begin DLL Section

Begin DLL indicates that Business Function Builder is building a particular DLL. For example, assume that the previous section begins with`*****CDIST*****`. A *Begin DLL* section appears for each DLL that you are building.

3.7.2.3 Compile Section

Before it build DLLs, Business Function Builder compiles the business functions in the DLLs first. The system displays a sequential list of each business function that the Business Function Builder attempts to compile. During the compilation process, these events might occur:

- Compiler Warning

  When a compiler warning occurs, JD Edwards EnterpriseOne Business Function Builder displays `warning C`XXXX (where XXXX is a number) and a brief description of the warning. To review information about the warning, search for the `C`XXXX value in Visual C++ online help. Warnings usually do not prevent the business function from compiling successfully. However, you can select the Warnings As Errors option in the Global Build form so that the business function will not build if any warnings occur.

- Compiler Error

When a compiler error occurs, JD Edwards EnterpriseOne Business Function Builder displays `error C`XXXX (where XXXX is a number) and a brief description of the error. To review extended information about the error, search for the `C`XXXX value in Visual C++ online help. Because errors prevent the business function from compiling successfully, you must resolve them.

3.7.2.4 Link Section

After Business Function Builder has compiled the business functions for a DLL, it links them. This linking process creates the .lib and .dll files for the DLL. During linking, these events might occur:

- Linker Warning

  When a linker warning occurs, JD Edwards EnterpriseOne Business Function Builder displays `warning LNK`XXXX (where XXXX is a number) and a brief description of the warning. To review information about the warning, search for the `LNK`XXXX value in the Visual C++ helps. Warnings usually do not prevent the business function from linking successfully. You can select the Warnings As Errors option in the Global Build form so that the DLL will not build if it has any warnings occur.

- Linker Error

  When a linker error occurs, JD Edwards EnterpriseOne Business Function Builder displays `error LNK`XXXX (where XXXX is a number) and a brief description of the error. To review extended information about the error, search for the `LNK`XXXX value in the Visual C++ helps. If a nonfatal error occurs, Business Function Builder still creates the DLL. However, JD Edwards EnterpriseOne Business Function Builder notes that the DLL was built with errors. If a fatal error occurs, JD Edwards EnterpriseOne Business Function Builder does not build the DLL.

3.7.2.5 Rebase Section

The Rebase Section displays information about rebasing. Rebase fine-tunes the performance of DLLs so that they load faster. Rebase does this by changing the desired load address for the DLL so that the system loader does not have to relocate the image. The system automatically reads the entire DLL and also updates fixes, debug information, checksum information, and time stamp values.

3.7.2.6 Summary Section

The Summary Section contains the most important information about the build. This section indicates whether the build is successful. The summary section begins with *****Build Finished*****. JD Edwards EnterpriseOne Business Function Builder also displays a summary report for each DLL that you attempted to build. This report includes this information:

- The number of warnings.
- The number of errors.
- Whether the DLL build is successful.

3.7.3 Building All Business Functions

You can use Build All to build all business functions. Build All performs the same operations as global link, and it recompiles all of the objects within each DLL. A system administrator usually runs Build All. Build All processes can take a long time. To run Build All, you must access BusBuild.

To build all business functions:

1. In Windows, run BusBuild.exe.

   Typically, this file is located in ..\B9\system\Bin32\.

2. In BusBuild, start the mass build by selecting Build, Build All.

3. Select one of these options for Build Mode:

   – **Debug**

     A build that includes debug information. After you perform a build, you can debug the built business function using the Visual C debugger.

   – **Optimize**

     A build that does not include debug information. Optimized builds generally cannot be debugged using the Visual C debugger.

   – **Performance Build**

     A build that is the same as an optimized build except that it includes information that helps developers measure the performance of business functions. Only JD Edwards developers should select this option.

4. Complete the Source Directory field.

   Use this field to specify where the business function source resides. Business function source includes all .c, .h, named event rules, and table event rules. Full packages usually have all business function sources. These are the options for location:

   – **Local**

     All business function source is on the local machine.

   – **Path Code**

     All business function source is in the path specified by the selected path code.

   – **Package**

     The All business function source is in the path specified by the selected package. If a package is built correctly, it typically contains all required business function sources. Generally, you should use **Package** for the location.

   – **Pick Directory**

     All business function source is stored in another directory on the file server. You specify the directory.

5. Complete the Foundation Directory field.

   Use this field to specify the foundation to use for this build. The foundation that you select is the foundation on which you expect these business functions to run. These are the options for this field:

   – **Local**

     The recommended foundation is the local JD Edwards EnterpriseOne foundation.

   – **Foundation**

     The foundation table lists all registered JD Edwards EnterpriseOne foundations. Select a foundation from this table.

   – **Pick Directory**

The JD Edwards EnterpriseOne foundation exists in a directory on the file server. You specify the directory. JD Edwards EnterpriseOne recommends this location.

6. Complete the Output Destination Directory field.

   Use this field to specify the location for the output of the build. The build output includes the file types: DLL, .LIB, .OBJ, and LOG. The location options are the same as those for **Source Directory**. Generally, you should select **Package** because it is a more stable snapshot of business function source.

7. Select any of these options:

   – Treat Warnings As Errors

     If you select this option, JD Edwards EnterpriseOne Business Function Builder does not build a business function if it encounters any warnings.

   – Clear Output Destination Before Build

     If you select this option, JD Edwards EnterpriseOne Business Function Builder deletes the contents of the bin32, lib32 and obj output directories before it builds all business functions.

   – Select Which DLLs to Build

     If you clear this option, JD Edwards EnterpriseOne Business Function Builder builds all DLLs. If you select this option, you can click the Select button and select which business function DLLs you want to build. Select this option if you want to build one or two DLLs. If you build only a subset of all DLLs, verify that the Clear Output Destination Before Build option is cleared.

   – Stop Level

     You can select the error level at which the build stops. You can ignore errors if you want to continue building despite them. You can specify that the build process stop if a DLL contains errors. You can stop on the first compile error.

   – Generate Missing Source Report

     If you select this option, v Business Function Builder generates a report in the work directory of the destination. This report is called NoSource.txt. It contains business function source file names that do not have a .c file but do have a record in the F9860 table. To resolve the information in this report, you can produce the correct .c file for the business function, or you can delete the source file from the F9860 table. It is recommended that you select this option.

   – Generate ER Source

     If you select this option, v Business Function Builder generates NER and table event rule source before building business functions.

   – Verify Check-in

     If you select this option, the system builds only objects checked in to a specified path code. A log file, Notchkdn.txt, is written to the same directory as Nosource.txt. Objects that are not checked in to the path code will be listed in this log and in Buildlog.txt.

Select the From RDB option to generate work from any path code. If this option is cleared, the business function builder assumes that the event rules source can be generated from the source directory specification files.

If you are troubleshooting a build initiated by **Package Build**, then the previous settings should already be set to the correct values. In this case, click Build to rebuild the problem DLLs.

Note: You can also run this build by selecting the Build BSFN option on in a package build.

3.7.4 Using the Utility Programs

The Tools menu contains several utility programs that assist in the build process. This table lists those utilities:

Utility	Purpose
Synchronize JDEBLC	You run the Synchronize JDEBLC program to reorganize JD Edwards EnterpriseOne business functions into new DLL groupings. This program synchronizes DLL field for the local JDEBLC parent specification table with the parent DLL in the F9860 table. Use this program with caution. You typically use this program only if you have manually dragged business function DLLs from a recent package build and you are experiencing failures in the business function load library.
Dumpbin	You run the Dumpbin program to verify whether a particular business function built successfully. This program displays all the business functions that were built into the selected consolidated DLL.
PDB (Program DeBug file) Scan	You receive a CVPACK fatal error when one of the object files that you are trying to link is incorrectly compiled with PDB information. To resolve this problem, you can use the PDB Scan to identify any object fields that were built with PDB information. Recompile any business functions that the PDB Scan reports.
Customize	You use Customize to add programs to the Tools menu. For example, you could add the programming tool and pass that tool a file name as a parameter when it opens.
Safety Check	You use Safety Check to check selected files (.c, .h or both) for: ■ global variables ■ static variables ■ extern declarations ■ non-"threadsafe" ANSI C APIs
Safety Check-Check All	You use Safety Check-Check All to check all files (.c, .h or both) in a directory for the same conditions as for Safety Check.

3.7.4.1 Resolving Errors with JDEBLC, Dumpbin, and PDB

You use JD Edwards EnterpriseOne Business Function Builder tools to help you resolve errors. If you notice any unresolved external errors during a business function build, the consolidated DLL still builds, and the software should run normally. However, it cannot execute any unresolved business function.

Use the dumpbin tool to verify that a particular business function is present in a consolidated DLL. If a business function is present, its name appears in the dumpbin output, followed by a nonzero number in parentheses.

Use the PDB scan to resolve the CVPACK fatal error. The CVPACK error occurs when the Business Function Builder attempts to link an object file that was built with PDB (Program DeBug file) information. The PDB scan finds the problem object file. You

must then recompile the problem object file on the machine with the JD Edwards EnterpriseOne Business Function Builder.

If a business function is compiled using Visual C++, it will not work properly. You can use PDB scan to identify any business functions that have been built outside of JD Edwards EnterpriseOne Business Function Builder. Use JD Edwards EnterpriseOne Business Function Builder to rebuild these functions so that they work properly.

If one of the DLLs is out of synch, you must rebuild it using the Build option. This generates a makefile and then relinks all the business functions within it.

The Synchronize JDEBLC option from the JD Edwards EnterpriseOne Business Function Builder Tools menu corrects any misplaced or incorrectly-built business functions. This option reviews the server DLLs and determines whether the local workstation specifications match those of the server. If they do not, then JD Edwards EnterpriseOne Business Function Builder will rebuild the business functions in the correct DLL on the server and relink them.

The Build Log contains these sections:

Section	Description
Build Header	This section defines the configuration for a specific build, including the source path, foundation path, and destination path.
Build Messages	This section displays the compile and link activity. During a compile, a line is output for each business function that was compiled Any compile errors are reported as error cxxxx. During the link part, business function builder outputs the text `Creating library ....` This text might be followed by linker warnings or errors.
Build Summary	The last section of the build summarizes the build for each DLL. This summary is in the form `x error(s), x warnings (y)`. The summary indicates the status of the build. If you have no warnings and no errors, then the build was successful. If the summary reports an error, search the log for the word *error* to determine the source of the error. Typical build errors are syntax errors and missing files.

3.7.4.2 Customizing the Tools Menu

This table lists the sections of the Customize menu option:

Menu Option	Usage
Menu Contents	Review all current tools menu customizations.
Menu Text	Enter the text to display in the menu.
Command	Enter the executable to run. You must supply a full path for any program that does not reside in `system\bin32` or that is not defined in Initial Directory.
Arguments	Specify any command line arguments to pass to the executable.
Initial Directory	Specify the initial directory that should be used by the executable, if it is not `system\bin32`.
Include in Build	Select to display output from the program as part of the build process. **Note:** This option is only valid and will only appear for Release 8.11 SP1 or later. If you are running an earlier version, this option is not available, and Safety Check does not run during build. You must, instead, run Safety Check manually from the menu.

Menu Option	Usage
Hide Window	Select to hide command windows. The functionality remains the same.

This table lists the buttons in the Customize menu option:

Button	Usage
Add	Select to enter new programs to appear in the pull-down menu.
Remove	Select to remove the selected item from the menu.
Move Up	Select to move the selected item up in the menu.
Move Down	Select to move the selected item down in the menu.
Ellipsis	Select to open a file or directory dialog so that you can browse for a file or directory.
Question Mark	Select to display a list of substitutions you can use as part of command line arguments. In our SafetyCheck example, one of the command line arguments is: `--F <source_file>`. By specifying `<source_file>`, you are telling SafetyCheck to use as its input file the selected source file. When BusBuild starts the build process, it can determine which file is being built and substitute that name in place of the text `<source_file>`.

3.7.4.3 Threadsafe Code

All BSFNs created for JD Edwards EnterpriseOne 8.11 Applications Release and earlier Applications releases are designed to run in a single-threaded environment. BSFNs designed for JD Edwards EnterpriseOne 8.11 SP1 Applications Release and later Applications releases that also run with JD Edwards EnterpriseOne Tools Release 8.96 and later Tools releases are designed to run in a multi-threaded environment. To be considered threadsafe, BSFNs cannot use:

- Global variables.
- Static variables.
- External declarations.
- Non-threadsafe ANSI C APIs.

Safety Check is a source code analysis tool that scans C source code and header files for non-threadsafe behaviors. Given a source or header file, Safety Check finds all instances of non-threadsafe code, returning line numbers and code fragments.

Several non-threadsafe APIs have a JD Edwards EnterpriseOne replacement. These replacement APIs have the same parameters as the non-threadsafe C APIs, except where noted. Most non-threadsafe APIs do not have a JD Edwards EnterpriseOne replacement. These APIs and their replacements do not necessarily have the same parameters. Use care when using these APIs.

This table lists the non-threadsafe C APIs for which SafetyCheck searches, the threadsafe standard C replacements, and the threadsafe JD Edwards EnterpriseOne replacements (if applicable):

Non-Threadsafe Standard C API	Threadsafe Standard C API	Threadsafe JD Edwards EnterpriseOne API
acltostr	**acltostr_r**	None

Non-Threadsafe Standard C API	Threadsafe Standard C API	Threadsafe JD Edwards EnterpriseOne API
asctime	**asctime_r**	**jdeJAsctime**
crypt	**crypt_r**	None
ctime	**ctime_r**	**jdeJCtime**
drand48	**drand48_r**	None
ecvt	**ecvt_r**	None
encrypt	**encrypt_r**	None
endgrent	**endgrent_r**	None
endhostent	**endhostent_r**	None
endnetent	**endnetent_r**	None
endprotoent	**endprotoent_r**	None
endpwent	**endpwent_r**	None
endservent	**endservent_r**	None
endspwent	**endspwent_r**	None
endusershell	**endusershell_r**	None
endutent	**endutent_r**	None
erand48	**erand48_r**	None
fcvt	**fcvt_r**	None
fgetgrent	**fgetgrent_r**	None
fgetpwent	**fgetpwent_r**	None
getdate	**getdate_r**	None
getdiskbyname	**getdiskbyname_r**	None
getgrent	**getgrent_r**	None
getgrgid	**getgrgid_r**	None
getgrnam	**getgrnam_r**	None
gethostbyaddr	**gethostbyaddr_r**	**jdeGetHostByAddr_r**
gethostbyname	**gethostbyname_r**	**jdeGetHostByName_r**
gethostent	**gethostent_r**	None
getlocale	**getlocale_r**	None
getlogin	**getlogin_r**	None
getnmtent	**getmntent_r**	None
getnetbyaddr	**getnetbyaddr_r**	None
getnetbyname	**getnetbyname_r**	None
getnetent	**getnetent_r**	None
getprotobyname	**getprotobyname_r**	**jdeGetProtoByName_r**
getprotobynumber	**getprotobynumber_r**	None
getprotoent	**getprotoent_r**	None
getpwent	**getpwent_r**	None

Non-Threadsafe Standard C API	Threadsafe Standard C API	Threadsafe JD Edwards EnterpriseOne API
getpwnam	**getpwnam_r**	None
getpwuid	**getpwuid_r**	None
getservbyname	**getservbyname_r**	None
getservbyport	**getservbyport_r**	None
getservent	**getservent_r**	None
getspwaid	**getspwaid_r**	None
getspwnam	**getspwnam_r**	None
getspwuid	**getspwuid_r**	None
getusershell	**getusershell_r**	None
getutent	**getutent_r**	None
getutid	**getutid_r**	None
getutline	**getutline_r**	None
gmtime	**gmtime_r**	**jdeGmtime**
inet_ntoa	**inet_ntoa_r**	**jde_inet_ntoa_r**
jrand48	**jrand48_r**	None
l64a	**l64a_r**	None
lcong48	**lcong48_r**	None
localtime	**localtime_r**	**jdeLocaltime** **Note:** The parameters changed on this due to the need to send a location to store the value. The standard C call stores it in a global static variable, which is not threadsafe.
lrand48	**lrand48_r**	None
ltoa	**ltoa_r**	None
ltostr	**ltostr_r**	None
mrand48	**mrand48_r**	None
nrand48	**nrand48_r**	None
ptsname	**ptsname_r**	None
pututline	**pututline_r**	None
rand	**rand_r**	**jdePPRand** **Note:** Must be used in conjunction with **jdePPSRand** to seed the random number generator correctly. Existing calls to **srand** should be replaced with **jdePPSRand.**
readdir	**readdir_r**	None
seed48	**seed48_r**	None
setgrent	**setgrent_r**	None
sethostent	**sethostent_r**	None
setkey	**setkey_r**	None
setlocale	**setlocale_r**	**jdeSetLocale**

Non-Threadsafe Standard C API	Threadsafe Standard C API	Threadsafe JD Edwards EnterpriseOne API
setnetent	**setnetent_r**	None
setprotoent	**setprotoent_r**	None
setpwent	**setpwent_r**	None
setservent	**setservent_r**	None
setspwent	**setspwent_r**	None
setusershell	**setusershell_r**	None
setutent	**setutent_r**	None
srand	**srand_r**	**jdePPSRand**
srand48	**srand48_r**	None
strerror	**strerror_r**	None
strtoacl	**strtoacl_r**	None
strtoaclpatt	**strtoaclpatt_r**	None
strtok	**strtok_r**	None
ttyname	**ttyname_r**	None
ultoa	**ultoa_r**	None
ultostr	**ultostr_r**	None
utmpname	**utmpname_r**	None
wcstok	**wcstok_r**	None

3.7.4.4 Safety Check Usage

During the course of development, there may be times when a non-threadsafe type of code must be used. You can mark source code with an explanation about why the non-threadsafe code exists. Safety Check will then display this information as part of its run. To mark source code with an exception, include a comment in this format: `/*_LRBF <comment text */`. The comment must begin with "`/*_LRBF`." The remainder of the comment can span multiple lines and include any other necessary text. The entire comment will print as part of Safety Check output.

You control Safety Check functionality through several options, at least one of which must be supplied. Multiple options are supported. Quotation marks are required only when the path specified contains spaces. For example, if the single C source file b1234.c is stored in the "c:\source" directory, you could call SafetyCheck in one of two ways: SafetyCheck --F c:\source\b1234.c or SafetyCheck --F "c:\source\b1234.c" However, if the same C source file is stored in the "c:\test files", you must enclose the path/filename in quotations: SafetyCheck --F "c:\test files\b1234.c"

Argument	Usage
--F<C source file>	Use to check a single C source file, for example, **--F** c:\test\b1234.c
--I<Header file>	Use to check a single header file, for example, **--I** c:\include\b1234.h

Argument	Usage
--FD<C source directory>	Use to check all C source files in a given directory, for example, --FD c:\my project\source. **Note:** Do not include a trailing slash as part of the directory argument.
--ID<Header file directory>	Use to check all header files in a given directory, for example **--ID** c:\my project\include **Note:** Do not include a trailing slash as part of the directory argument.

--P<Project file>	Use to create a text file that contains a list of files, each of which will be scanned by Safety Check. The project file should contain multiple lines of the form: **SOURCE=**<fully qualified file name>
	
	Note: Do not use quotation marks in the project file.
	
	For example, a project file that specifies three files to scan could look like this:
	
	`SOURCE=c:\my project\source\b1111.c`
	
	`SOURCE=c:\my project\source\b2222.c`
	
	`SOURCE=c:\my project\include\main.h`

--csv	Use to produce output in a comma-delimited format. The output will contain these elements: ■ File (the fully qualified file name) ■ Line (the line number of the erroneous code) ■ Global (1 if a global was found, 0 if not.) ■ Static (1 if a static was found, 0 if not.) ■ Extern (1 if an external declaration was found, 0 if not.) ■ API (1 if a non-threadsafe API was found, 0 if not.) ■ BraceMismatch (1 if scanning could not completedue to a brace mismatch) ■ Exception (1 if an exception comment was found, 0 if not.) ■ CouldNotOpen (1 if the file could not be opened, 0 if it could.) ■ NotCSource (1 if the file name did not end in either ".c" or ".h") ■ C++Comment (1 if a C++ style comment was found) ■ CapInclude (1 if a capital letter was used in a #include) ■ LastChar (1 if the last character was not a new line character) ■ CommentInComment (1 if a comment was found inside a comment)
--X	Select to print a warning message when a file to check is specified that does not end in "c" or "h". By default, these warning messages are hidden.

3.7.4.5 Safety Check Output

A "clean" Safety Check run will produce output of this format:

```
---------------- SafetyCheck Started ---------------

Scanning d:\safetychecktestrun\source\b03b0011.c...
```

```
---------------------- Done ----------------------

1 Files Processed 0 Errors 0 Warnings
```

"Files processed" indicates how many files were scanned. "Errors" reports the number of file-based errors encountered. "Warnings" reports the number of problems found while scanning the specified files.

A "dirty" Safety Check run will produce output of this format:

```
---------------- SafetyCheck Started ---------------

Scanning d:\safetychecktestrun\source\b03b0011.c...

d:\safetychecktestrun\source\b03b0011.c(186): Global variable found

int iGlobal = 0;

---------------------- Done ----------------------

1 Files Processed 0 Errors 1 Warnings
```

In this case, the output indicates:

- A problem was found in d:\safetychecktestrun\source\b03b0011.c
- The problem occurred on line 186.
- The problem found was the presence of a global variable.
- The section of code that caused the problem is "`int iGlobal = 0;`"

Note that the global variable was specified as a "Warning" and not an "Error".

3.7.4.6 Safety Check Limitations

Following are limitations for safety check:

1. Safety Check is a static code analysis tool that does not perform preprocessing of source code. Therefore, macro substitutions may introduce non-threadsafe behaviors that cannot be detected by Safety Check.
2. Safety Check does not know which compile-time flags may be set. Problems will occur in code that looks like this because the number of open braces does not match the number of close braces:

   ```
   int FunctionOne(int i) { if (i == 0) #ifdef FLAG1 { ++i; #else { --i;
   #endif } }
   ```

3. Non-threadsafe code may still exist even though Safety Check reports no warnings. Safety Check is looking for the presence of only four specific code elements (globals, variables, externs and non-threadsafe ANSI C APIs). Do not rely solely on a "clean" run of Safety Check as the only test of whether the code is threadsafe.

3.7.5 Understanding Business Function Processing Failovers

In some instances in which a business function fails to process correctly, the software can attempt to recover and reprocess the transaction. The system recognizes two principle failure states: process failure and system failure.

A process failure occurs when a jdenet_k process aborts abnormally. For a process failure, the software server processing launches a new jdenet_k process and continues processing.

A system failure occurs when all the server processing fails, the machine itself is down, or the client cannot reach the server because of network problems. For a system failure, business function processing must be rerouted either to a secondary server or to the local client. The system uses this process to attempt to recover from this state:

- When the call to the server fails, the system attempts to reconnect to the server.
- If reconnect succeeds and no cache exists, the system reruns the business function on the server.

  If a cache does exist, the system forces the user out of the application.
- If reconnect fails and no cache exists, the system switches to a secondary server or to the local client.

  If a cache does exist, the system forces the user out of the application.

After one module switches, all subsequent modules switch to the new location.

3.8 Working with Business Function Documentation

This section provides an overview of business function documentation, and discusses how to:

- Create business function documentation.
- View documentation from the Business Function Documentation Viewer.

3.8.1 Understanding Business Function Documentation

Business function documentation explains what individual business functions do and how they should be used. The documentation for a business function should include this type of information:

- Purpose.
- Parameters (the data structure used).
- Descriptions for each parameter that indicate required input and output, and explain return values.
- Related tables (the table accessed).
- Related business functions (business functions called from within the function itself).
- Special handling instructions.

You use Business Function Design and Data Structure Design to document the business functions.

3.8.2 Creating Business Function Documentation

You can create business function documentation for several levels, including these:

- Business Function Notes

  Documentation for the specific business function that you are using.
- Data Structure Notes

  Notes about the data structure for the business function.
- Parameter Notes

Notes about the actual parameters in the data structure.

Generating business function documentation provides you with an online list of information about business functions that you can view through the Business Function Documentation Viewer (P98ABSFN). Typically, the system administrator performs this task because generating the business function documentation for all business functions takes considerable time. If you create new business function documentation, you need to regenerate the business function documentation for that business function only.

Run UBE R98ABSFN, batch version XJDE0001 to generate all business function documentation. The system creates a hypertext markup language (HTML) link for each business function for which you generated documentation. It also creates an Index HTML file. These HTML files appear in the output queue directory.

3.8.3 Viewing Documentation from Business Function Documentation Viewer

You can use Business Function Documentation Viewer to view documentation for all business functions or selected business functions. After you generate the report, use the Business Function Documentation Viewer (P98ABSFN) to display the information. It is suggested that you use this method to view business function documentation.

The Business Function Documentation form contains the HTML index that you generated. To view the entire index or select specific functions, click the appropriate letter in the index. Double-click a business function to view documentation that is specific to that function.

The media object loads the HTML index of the business functions based on a media object queue. In the media object queue table, a queue named Business Function Doc is defined.

This queue must point to the directory in which the business function HTML files are located. The system administrator usually generates the documentation for all business functions. Because the generation process places the documentation files in the local directory, the administrator must then copy the files to a central directory on the deployment server. The files must be copied to the media object queue for media object business function notes. If you are using the standalone version of the software, this path is usually the output directory from the Network Queue Settings section of the jde.ini file. If this entry is not in the jde.ini file, it is in the print queue directory in the JD Edwards EnterpriseOne software directory.

4

Understanding Record Locking

This chapter contains the following topics:

4.1 Record Locking

JD Edwards EnterpriseOne does not implement any record-locking techniques. It relies on the native locking strategy of the vendor database management system.

In specific situations, the vendor database does not automatically lock as needed. In these situations, you can instruct JD Edwards EnterpriseOne to control record locking. For example, you can mandate record locking on the Next Numbers table to ensure the integrity of the Next Numbers feature.

You can lock JD Edwards EnterpriseOne records using one of the following methods:

- Optimistic locking

  Use optimistic locking (sometimes referred to as record change detection) to prevent a user from updating a record if it has changed between the time the user inquired on the record and the time user updates the record.

- Pessimistic locking

  Use pessimistic locking to prevent attempts to update the same record at the same time by different applications or users. The record is locked before it is updated.

4.2 Optimistic Locking

You can set optimistic locking in the workstation jde.ini file. This type of database locking prevents a user from updating a record that changed since the user has inquired about it. If the record has changed, the user must select the record again and then make the change. This feature is available for business functions, table I/O, and Named Event Rules.

For example, assume that two users are working in the Address Book application The following table illustrates the optimistic locking process:

Time	Action
10:00	User A selects Address Book record 1001 to inspect it.
10:05	User B selects Address Book record 1001 to inspect it. Both users now have Address Book record 1001 open.
10:10	User B updates a field in Address Book record 1001 and clicks OK. JD Edwards EnterpriseOne updates Address Book record 1001 with the information entered by User B.
10:15	User A updates a field in Address Book record 1001 and clicks OK. JD Edwards EnterpriseOne does not update Address Book record 1001, and the system displays a message informing User A that the record has changed during the time that User A was viewing it. For User A to change the record, User A must re-select it and then update it.

When the system detects that a record change has occurred, it displays a message indicating that the record has been changed since it was retrieved.

4.3 Pessimistic Locking

Pessimistic locking is sometimes referred to as record locking. You can use pessimistic locking to prevent multiple users or applications from updating the same record at the same time. For example, suppose a user enters a transaction that uses Next Numbers. When the user clicks OK, the Next Numbers feature selects the appropriate Next Numbers record, verifies that this number is not already in the transaction file, and then updates the Next Numbers record by incrementing the number. If another process tries to access the same Next Numbers record before the first process has successfully updated the record, the Next Numbers function waits until the record is unlocked and then completes the second process.

Pessimistic locking in JD Edwards EnterpriseOne is implemented by calling published JDEBase APIs. When you use pessimistic locking, you should consider the time required to select and update a record because the record is locked until the update is complete. Transaction processing uses a special set of locking APIs. A locked record might or might not be part of a transaction. Record locking APIs are independent of the transaction and its boundaries. They always lock, regardless of whether you are in manual or auto commit mode.

Records that are updated using pessimistic locking APIs (such as JDB_FetchForUpdate or JDB_UpdateCurrent) within a transaction boundary are locked from the time the record is selected for update until the commit or rollback occurs. Records within the transaction boundary that are updated without using pessimistic locking APIs are locked from the time of the update until the commit or rollback occurs. This is also true if you use a business function to define and activate transaction processing.

4.3.1 Using Pessimistic Locking Within a Transaction Boundary

You might need to use pessimistic locking in conjunction with transaction processing. For example, if you want the system to lock records between the read operation and the update, you must use pessimistic locking.

4.3.2 Business Functions and Pessimistic Locking

You might want to use pessimistic locking in a business function if the business function updates a table. The table being updated should have a high potential for record contention with another user or job. Remember that you should lock records for as short a time as possible. Ensure that the select or fetch for an update occurs as closely to the update as possible.

5

Debugging Business Functions

This chapter contains the following topics:

5.1 Debugging

Debugging is the method you use to determine the state of your program at any point of execution. Use debugging to help you solve problems and to test and confirm program execution.

Use a debugger to stop program execution so you can see the state of the program at a specific point. This enables you to view the values of input parameters, output parameters, and variables at the specified point. When program execution is stopped, you can review the code line-by-line to check such issues as flow of execution and data integrity.

You use the Visual C++ Debugger to debug C business functions.

5.2 Debugging Strategies

You can use several strategies to make debugging faster and easier. Begin by observing the nature of the problem.

5.2.1 Is the Program Ending Unexpectedly?

If the program is ending unexpectedly, the cause is likely an unhandled exception. An unhandled exception is a failure to handle memory correctly. It is an easy problem to track down if it is happening in the same place: simply set breakpoints at strategic points throughout the code and run the program until you find the problem.

If other objects are missing, termination is more abrupt. Remember to transfer all Media Object (also called Generic Text) objects correctly. If an application has a Row exit to an application that does not exist, an unhandled exception in the program occurs immediately.

Termination of the program is more abrupt and less helpful when other kinds of objects are missing. You must review all of the pieces of the application to verify that they are all present and correctly built. A common error is to overlook media objects. If you cannot enter the program at all, a missing object is most likely the problem.

Ensure that the program is terminating in the same place. If the program is failing to restore memory after its use, the program might eventually have insufficient memory to run. If so, you must reboot the workstation to restore memory.

5.2.2 Is the Output of the Program Incorrect?

Incorrect program output typically indicates a flaw within the logic of the code. To help find the error:

- Set a breakpoint in the code prior to the point where the bad output is produced.
- Step through the ER line by line, while monitoring the values of relevant ER variables.

  At some point, a variable will probably take on an erroneous value that subsequently produces incorrect output.
- If that point occurs before your breakpoint, set another breakpoint earlier in the code and restart the application.
- Continue this process until you find the statement that is causing the wrong value to be assigned to the variable.

5.2.3 Where Else Could the Problem Be Coming From?

Spend some time thinking about where the source of the problem might be. If you don't know which ER event is causing an error, try to isolate it. For example, you might be able to temporarily disable the ER one event at a time to see if the error still happens. You can try to repeat the processing of a single event by doing unnatural actions in the GUI, like toggling up and down between grid rows to force the execution of the **Row Is Exited** event. There are no predefined debugging strategies that will work in any given situation. Be creative and be persistent, until you narrow down the problem to its source.

5.3 Debug Logs

You can output to a file a log of SQL statements and events by changing the line in the jde.ini file under `[DEBUG]` from `Output = NONE` to `Output = FILE`, as in the following sample. This is a useful debugging tool when you have narrowed a problem to a specific issue involving the JDEDB APIs.

```
[DEBUG]
TAMMULTIUSERON=0
Output=FILE
ServerLog=0
LEVEL=BSFN,EVENTS
DebugFile=c:\jdedebug.log
JobFile=c:\jde.log
Frequency=10000
RepTrace=0
```

You can set breakpoints and examine the code.

5.4 Debugging Business Functions with Microsoft Visual C++

This section provides an overview of the Microsoft Visual C++ debugger and describes how to:

- Debug business functions attached to interactive applications.

- Use SQL log tracing.
- Use debug tracing.

5.4.1 Understanding the Visual C++ Debugger

You can use Microsoft Visual C++ to debug business functions that are written in C. You can debug business functions that are attached to interactive applications or to batch applications. The business function must be configured to run locally.

If you are debugging ER for business functions and C business functions, you can use the JD Edwards EnterpriseOne debugger and the Visual C++ debugger together. Follow the process until you log into JD Edwards EnterpriseOne. At that point, follow the steps for the JD Edwards EnterpriseOne debugger. Program execution stops if C code is accessed. You can then use Visual C++ to continue debugging. This method is useful if you are trying to locate a problem and are not sure whether the problem is in a C business function or in the application that calls the business function.

You must use the Microsoft Visual C++ Debugger to debug business functions that were written with the Event Rules scripting language and then interpreted as C code, or that were originally written in C . You can run the entire JD Edwards EnterpriseOne system through the Visual C++ debugger (that is, you can start the activeConsole.exe or JD Edwards Solution Explorer file from within the Visual C++ Debugger). This enables you to step out of the tool application code into the business functions that are called in the ER.

You can use the debugger to debug a C program and interactively stop and start it as needed. During debugging, you can check specific values of variables and parameters to determine whether a program is running correctly. You can also step through the code to see what code is actually being executed.

The debug commands are listed in the Debug menu. You can customize the tool bar to contain debug buttons, which you can use instead of the menu.

The Visual C++ has many features in the Debug menu. The Visual C++ debugger helps you efficiently solve real-world problems.

5.4.1.1 The Go Command

You can run a program using the Go command from the Debug menu. The program runs until completion unless you set up breakpoints.

5.4.1.2 The Step Command

The Step command is available on the Debug menu and executes the current line of code. When the line of code has been executed, the yellow arrow cursor appears on the next line of code to be executed.

5.4.1.3 The Step Into Command

You can access the Step Into command from the Debug menu. Use this command when the current line of code contains a function call. The debugger steps into the function so that it can be debugged line by line. When the function is complete, the debugger returns to the next line of code after the function call in the calling routine. If the source code of the function to be stepped into does not exist on the workstation, the debugger skips over the line of code as though the Step command was used.

Stepping into a standard C function takes you into the function, which you might not want to do. If so, use the Step Over command to skip those functions.

5.4.1.4 Setting Breakpoints

You use breakpoints to run the program until it reaches a certain line of code. If a breakpoint is set, the Go command runs the program until it encounters that line of code.

You can set a breakpoint by placing the cursor anywhere on the line of code. When you select Debug, Breakpoints, a red octagon appears to the left of the line of code where the breakpoint is set. When the program is run, all lines of code up to the breakpoint are executed. To continue execution after the breakpoint, you can use Step, Step Into, or Go .

5.4.1.5 Using Watch

You can use Watch to inspect what values variables are set to. To use Watch, click the item that you want to watch and drag it to the Watch window.

5.4.1.6 Locals Window

All local variables and parameters to a function are listed with their data types and values in the Locals window. You can modify the values of all items in the Locals window during debugging. This is useful if you are debugging infinite loops.

5.4.2 Understanding Visual C++ Debugger Tracing Utilities

Visual C++ has two tracing utilities that you might find valuable: SQL Log Tracing and debug tracing. You can use SQL Log Tracing to help you determine the exact SQL statement that is generated and sent to the database.

5.4.3 Debugging Business Functions Attached to Interactive Applications

To debug a business function attached to an interactive application:

1. Close the application.

   The application must be closed to debug in Visual C++.
2. Open Visual C++ and verify that all work spaces have been closed.
3. Select File, Open.
4. Select List Files of Type to accept executables (.exe).
5. Select activConsole.exe on path \b9\System\bin32 and click the OK button.

   The system creates a project work space.
6. Select Project, Settings.
7. Click the Debug tab.
8. In the Category list, select Additional DLLs.
9. Click the Browse button to select the CALLBSFN.dll (which must be built in debug mode) or other appropriate DLL on path \b9\path\bin32, where path varies, depending on the path code.
10. Click the OK button.
11. Select the .h and .c files for the source that you want to debug from and then select File, Open.
12. To set breakpoints in the code, select Edit, Breakpoints.

    If this message appears, click the OK button:

```
cannot open *.pdb
```

If a message appears notifying you that breakpoints have been moved to the next valid lines, a source code and object mismatch might exist, and you might need to rebuild the business function.

13. Select Build, Start Debug, Go.

    The JD Edwards EnterpriseOne sign-in window appears.

14. Sign in to the application as you normally would sign in.

15. Run the application.

    When the application reaches the business function in debug, the debugger opens or displays the C code in Visual C so that you can step through it.

5.4.4 Using SQL Log Tracing

This task is useful only for ODBC connections.

To use SQL Log tracing:

1. From the Control Panel on the workstation, select Administrative Tools, and then Data Sources (ODBC).
2. Select the 32 bit ODBC driver, and then click the Tracing tab.
3. Specify when you want the system to trace.
4. Specify the log output path in the Log file Path.

5.4.5 Using Debug Tracing

To use debug tracing:

1. In the jde.ini file under `[DEBUG]`, set `Output=FILE`.
2. Change the value for `Level=` to suit the specific debugging needs.

   Possible values for Level are contained in the comment line following the `Level=` line. Any combination is acceptable. Use commas to separate values.

Glossary

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.

checksum

A fixed-size datum computed from an arbitrary block of digital data for the purpose of detecting accidental errors that may have been introduced during its transmission or storage. JD Edwards EnterpriseOne uses the checksum to verify the integrity of packages that have been downloaded by recomputing the checksum of the downloaded package and comparing it with the checksum of the original package. The procedure that yields the checksum from the data is called a checksum function or checksum algorithm. JD Edwards EnterpriseOne uses the MD5 and STA-1 checksum algorithms.

deployment server

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

driver manager

The JDBC class that manages multiple registered JDBC drivers and dispatches connection initialization requests to them. The Java driver manager class is java.sql.DriverManager.

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.

Index

P

R

S

T

U

V

W

X