# AGILE E6.0.2
# DEVELOPER'S RELEASE NOTES

July 18, 2006

## Contents

Conceptual Product Structure (CPS)

Multi Access Rights

Call 215973

EPQ

- Oracle
- MS-SQL

# I18N Limited Support

## What is I18N and L10N?

- Internationalization (I18N):
  Internationalization is the process of developing a software product that core design does not make assumptions based on a locale. It potentially handles all targeted linguistic and cultural variations (such as text orientation, date/time format, currency, accented and double-byte characters, sorting, etc.) within a single code base. Isolating all message strings in text files is another necessary step to prepare a product for localization.

- Localization (L10N):
  Localization means taking an internationalized product and customizing it for a specific market. This includes translating the software strings, rearranging the UI components to preserve the original look and feel after translation, customizing the formats (such as date/time, paper size, etc.), the defaults, and even the logic depending on the targeted market. Such a customization is possible only if the application is properly internationalized; otherwise, the L10N team faces a challenge whose significance depends on the application and the language version.

Starting with the e6.0.2 release, we want to improve the I18N capabilties of our product. To do this, we need to enhance our database scheme, the C server and the clients.

The primary goal for e6.0.2 is to allow customers to include characters other than Latin1 in their multi-language data. This is a must-have when targetting the asian and east-european market; and it is also a must-have when it comes to integration or convergence with Agile 9.

## Overview

In e6.0.2, NCHAR columns with a Unicode charset are used to represent the non-Latin multi-language data in the ORACLE and MSSQL databases. This requires changes in the data model and the application server, but it has the benefit that all software components accessing the database directly using JDBC (or similar drivers) see the correct data as long as they support NCHARs.

### How to enable I18N

The first step is to determine which of the ten languages in the DataView repository should be capable of storing non-latin characters. This should be languages that are not yet in use, otherwise there might be migration issues.

For all of these languages, the new configuration column T_LANGUAGE.C_ENCODING has to be set to UTF-8.

- Use SQLPlus (or ISQL for SQL server) to change the entry in T_LANGUAGE.C_ENCODING to UTF-8:

```
SQL> UPDATE T_LANGUAGE SET C_ENCODING = 'UTF-8' WHERE C_SIGN = 'FRA';
SQL> COMMIT;
```

The national charset used in the database does not matter here, always enter UTF-8.

All multi-language database columns that represent one of the encoded languages must be converted to the NVARCHAR2 data type (ORACLE). Please contact SDO to get support for these step, they will support you with SQL scripts.

It might also be necessary to adapt column types of stored procedures from VARCHAR2 to NVARCHAR2 if they are hard coded.

The ChangeManagement ChangeLog tab is an example for such a procedure: it returns multi-language values (for instance EDB_TYPE and EDB_CHANGE_COMMENT).

The application server (DataView) treats all multi-language fields of the languages with encoding set to UTF-8 as encoded fields. Several new DataView functions have been added to deal with such fields (see DAL section in this document) .

New encoded fields added by the customizer are automatically created with the correct database type when *Create table* is executed. Existing database columns that represent encoded fields are automatically converted to the correct database type.

- Set the value of the System Configuration Parameter EDB-CHR-ENC-JVM to UTF-8.
  This tells Java and ECI clients which encoding they should use; UTF-8 allows them to use all available UniCode 2.0 characters.
- To be able to display non-Latin characters, you need a font that contains the characters needed for the respective languages. For instance, Microsoft Office comes with a universal Unicode font named *Arial Unicode MS* (**ARIALUNI.TTF**).

Add this font to the JavaClient setup:

1. Open Tools->Options->User interface and Connections
2. Go to the Mask settings and change the "fieldFont" entry using the font chooser. Select your Unicode font. Click OK to activate your changes.
3. Open your test mask with the French field and insert a new record.

This can also be pre-defined in the JavaClient default property file (jacc.defaults). For instance: to configure *Arial Unicode MS* with letter size 11, add this to the property file:

```
CommonMaskProperty.fieldFont=Arial Unicode MS,0,11
```

When using the WebClient, your Web Browser has to support the UTF-8 encoding, and it also needs a respective Unicode font.

## Installation Notes

### DataView Client

The DataView client needs a new registry entry name EP_CLIENT_CHARSET. It contains the canonical ICU name of the codepage used by the client. The DataView client always seems to use the Windows codepage Cp1252, which has the ICU name ibm-5348_P100-1997.

This client codepage is read by the server, but only used if server side character conversion has been activated (see 'e6 server' below).

### JavaClient

The JavaClient (and all other Java ECI clients) look into the configuration parameter EDB-CHR-ENC-JVM. For a standard environment, this is currently set to ISO-8859-15.
If NLS_LANG is set to AMERICAN_AMERICA.WE8MSWIN1252 (see below), this parameter should be set to windows-1252.

For an I18N environment it *must* always be set to UTF-8, regardless of the NLS_LANG setting.

Each user in an I18N environment needs to configure a proper font that is able to display non-Latin characters.

### WebClient

Same as JavaClient. Addtionally, the encodings defined in the webplm.properties file need to be adapted to use UTF-8 instead of ISO-8859-x in an I18N environment.

### JBoss

The Java property oracle.jdbc.defaultNChar must be set to "true" in an I18N environment that accesses an Oracle database. This is currently done in the run.bat/run.sh scripts and in the JBoss wrapper configuration BS_wrapper.conf It might be necessary to set this property only for I18N environments, because there are still some issues with some special characters like the '€'.

### e6 Server

Three new configuration entries exist to control how characters are mapped from the client to the database charset:

- If *EDB-EPQ-USE-CHAR-CONVERSION* is active the EPQ performs the clientside character set conversion.
- If *EDB-EPQ-USE-SUBST-CHARACTER* is set, invalid characters will be replaced with a substitution character, otherwise an error message is issued and the conversion is cancelled.

- If *EDB-EPQ-SUBST-CHARACTER* is set, its contents is used as substitution character, otherwise ICU's default substitution character (usually 0x1a) is used.

In the standard dump, the character conversion is deactivated. This is the configuration needed when upgraded dumps are used, because these dumps may already contain illegal characters (like the '€').

**Oracle Database**

- For upgraded environments, NLS_LANG needs to stay untouched (most likely it is AMERICAN_AMERICA.WE8ISO8859P15).
  In such an environment, no character conversion is done and the bytes are written to the database as they are.

- For new environments, NLS_LANG can be set to AMERICAN_AMERICA.WE8MSWIN1252.
  In such an environment, the character conversion is done by the Oracle library (which is the preferred method).

**MS SQL Server**

No special configuration necessary.

The character conversion from client codepage to database codepage is done by the e6 server.

## ECI Clients

ECI clients will get the standard 8-bit data unless they call eci_set_enc UTF-8 for *each connection* to the server. This ensures that unmodified clients do not get unsupported data from the server.

Two things happen when calling eci_set_enc UTF-8:

- The server will switch the data encoding to UTF-8 and will send all characters in UTF-8 encoding

- The server expects that all incoming data is in UTF-8, too!

The encoding to use can be read from the configuration parameter EDB-CHR-ENC-JVM. However, the client may decide to use UTF-8 even if the server is not setup for I18N.

The server currently supports only two encoding names:

- The standard 8-bit encoding used by the server.
  This should match the encoding name as specified in the EDB-CHR-ENC-JVM parameter.
  However, if an ECI client calls eci_set_enc with an unsupported encoding, an error code is returned and the ECI client can call eci_get_enc to get the name of the encoding used by the server.

- The standard I18N encoding: UTF-8
  This is always supported, even if the server is not configured for I18N.

Character conversion only takes place if UTF-8 is used. If the client does not use UTF-8, it has to use the encoding returned by eci_get_enc.

### Java ECI Clients

When using the Java ECI, an ECI client can set the encoding in the parameter object - either EciConParam or EciClientParam. The ECI connection implementation calls eci_set_enc automatically.

The encoding can be changed by calling setEncoding at any time, but be aware of the restrictions listed above.

UTF-8 encoding and decoding is done by the Java ECI.

When using the JET layer (i.e. the AxalantRepository class), everything is done automatically. The encoding used by the JET layer is read from EDB-CHR-ENC-JVM. As long as you call AxalantRepository.callEci() for all direct calls to the server, you don't have to worry about the encoding.

### C/C++ ECI Clients

The C ECI does not have parameter objects for the connection, so the client has to call eci_set_enc after opening a connection with eci_connect.

UTF-8 encoding and decoding must be done by the ECI API user. The ECI API does not do any character conversion at all.

To support the C developer, the ICU libraries have been added to the third party libraries shipped in $ep_root/ext/bin. The header files are at $ep_root/ext/inc/icu.

More information about the ICU can be found at this web address: International Components for Unicode (ICU).

## Limitations in this Release

### Conceptual Limitations

- The customizer has to configure which languages support non-Latin characters.
  This is configured in the new column T_LANGUAGE.C_ENCODING.
  This release only supports UTF-8 as encoding, an empty encoding column is interpreted as standard 8-bit encoding as returned by the EPQ.
  Currently, there is no configuration UI besides SQLPlus and ISQL.

- Only languages that are not yet in use by the customer may be configured to use encoded fields, we do not support migration of existing data.

- It is not supported to set one of these languages (see previous configuration step) as the UI language (or system language in DataView terminology).

- Only multi-language fields using one of the encoded languages can store non-Latin characters.
  These fields are called *encoded fields* in the rest of this document.
  We do not allow to use non-Latin characters in standard text fields because these are used as ID's, file names and so on.

- If the data stored in multi-lingual fields is used to generate field or mask titles, the user will see the raw UTF-8 data in the title. UI elements like titles are not yet capable of using non-Latin characters.

- Only the Java Client, the Web Client and the Workflow Editor will be able to display and modify encoded fields properly.
  In the Windows-Client these fields are displayed in UTF8 syntax (not human readable!) and are read-only to not tamper the value.

### Coding Restrictions

- Application code must not copy the content of an encoded field into a standard text field and vice versa.
  Doing otherwise may result in invalid multi-byte sequences. Such code needs to use for instance the ICU library to be on the safe side.

- Application code must not concatenate field content of a standard text field with the content of an encoded multi-language field.
  Doing otherwise may result in invalid multi-byte sequences. Such code needs to use for instance the ICU library to be on the safe side.

- Application code that parses encoded fields must be adapted to use the ICU library. Otherwise it might corrupt multi-byte sequences.

- LogiView restrictions
  The restrictions listed above apply to all kind of code running in the server, which includes LogiView.
  LogiView code should not concatenate or copy text from encoded fields to encoded fields or vice versa.
  We plan to add an I18N LogiView API in the next release.

### Customization Restrictions

Field menus that copy data from an encoded field into a standard text field need to be adapted: we recommend to use the Lookup-Table module (xlut userexits) instead of such menus:

- Copying a multi-language field into a non-multilangugae field is never a good idea. The LUT allows you to present a human readable multi-language text in the menu, and to store the ID assigned to this menu entry into the target field.

- Existing field menus that write multi-language data into multi-language fieldes work fine.

### Components that do not support Encoded Fields

This is a list of components where we identified problems with encoded fields:

- DataView Client: Shows raw UTF-8 data and might truncate multi-byte sequences.
- External Mail Client: Shows raw UTF-8 data
- IEF: Might truncate multi-byte sequences.
- RPG: Shows raw UTF-8 data and might truncate multi-byte sequences.
- PrintStudio: Shows raw UTF-8 data and might truncate multi-byte sequences.
- SML (Classification):
  The SML - especially the Attribute Pool - allows to enter non-Latin characters into encoded fields (which is fine), but this data is then used to create field titles.
  This will result in raw UTF-8 data written into the field titles. So in this release, we recommend to use only latin characters for attribute name and code.

## Activate I18N Features in Agile e6.0.2

### Requirements

- Agile e6.0.2 installation (at least WIP41).
- A font capable of displaying the characters you want to see.
  For instance, Microsoft Office comes with a universal Unicode font named *Arial Unicode MS* (**ARIALUNI.TTF**).

### ECI Clients

ECI clients will get the standard 8-bit ISO data unless they call eci_set_enc UTF-8 for *each connection* to the server. This ensures that unmodified clients do not get unsupported data from the server.

Two things happen when calling eci_set_enc UTF-8:

The server will switch the data encoding to UTF-8 and will send all characters in UTF-8 encoding

The server expects that all incoming data is in UTF-8, too!

The encoding to use can be read from the configuration parameter EDB-CHR-ENC-JVM. However, the client may decide to use UTF-8 even if the server is not setup for I18N.

The server currently supports only these encodings:

- ISO8859-1
- ISO8859-15
- UTF-8

Character conversion currently only takes place if UTF-8 is used. If the client does not use UTF-8, it has to use the encoding defined in EDB-CHR-ENC-JVM (by the way: this is also true for old server versions).

### Java ECI Clients

When using the Java ECI, an ECI client can set the encoding in the parameter object - either EciConParam or EciClientParam. The ECI connection implementation calls eci_set_enc automatically.

The encoding can be changed by calling setEncoding at any time, but be aware of the restrictions listed above.

UTF-8 encoding and decoding is done by the Java ECI.

When using the JET layer (i.e. the AxalantRepository class), everything is done automatically. The encoding used by the JET layer is read from EDB-CHR-ENC-JVM. As long as you call AxalantRepository.callEci() for all direct calls to the server, you don't have to worry about the encoding.

### C/C++ ECI Clients

The C ECI does not have parameter objects for the connection, so the client has to call eci_set_enc after opening a connection with eci_connect.

UTF-8 encoding and decoding must be done by the ECI API user. The ECI API does not do any character conversion at all.

To support the C developer, the ICU libraries have been added to the third party libraries shipped in $ep_root/ext/bin. The header files are at $ep_root/ext/inc/icu.

More information about the ICU can be found at this web address: International Components for Unicode (ICU).

## Basic I18N Tests

Configure your test environment to enable I18N support.

Choose one entity of your choice (should have at least one multi-lingual field, the more the better).
Ensure that at least one of the multi-lingual fields has lng_cop_dat as post-field userexit.

Do basic input tests in the form and list:

- ❑ The first test is simple:
  Start the DataView client and try to type something into an I18N field: this should not work, because these fields are set to read-only for this client.

Now start the Java and/or the Web client and perform these tests:

- Add "normal" characters into all fields.
  This should work. If lng_cop_dat is assigned to a field, the text should be copied to the other languages.

- Add "special" characters (Japanese, Cyrillic) into all fields:

  - For standard non-I18N fields, you should get an error message.
    The error message should appear when you try to leave the field.

  - For multi-lingual fields, you should be able to store the field content.
    However, the other non-I18N languages should be empty, even if lng_cop_dat is triggered.

- Check that you are able to store the number of characters allowed for each field.
  Remark: Due to a DataView limitation, the maximum number of characters for an I18N field is limited to 666 (no offense intended):

  DataView cannot store more than 2000 bytes per field.
  Because we are using UTF-8 to represent the characters, we can store at most 2000/3 = 666 characters in each field.

- Perform field queries with I18N fields using the data entered in the above steps:

  - Search for complete I18N text

  - Search using the single wildcard sign '?'

  - Search using the multi wildcard sign '%'

Connect as manager user using the DataView client

  - Use the ASCII loader to export your test data
    Delete your test data
    Reimport the loader file and check the content

  - Use the Binary loader to export your test data
    Delete your test data
    Reimport the loader file and check the content

- Setup Lightweight Reporting and try to print your test data

# DAL

## New userexit mas_set_mlf_fld

Userexit to set the multi-language flag of a field.

Useful to set a calculatory field to multi-lingual, which is needed to support I18N content in calculatory fields. To work properly, the field name has to have to suffix of the desired language, because DataView uses the last three characters of the field name to distinguish the language of the field.

This userexit will most likely be used by pre-mask triggers that add calculatory fields that should be I18N capable. The userexit only works on the in-memory instance of a mask, it does not change repository data.

Arguments:

cpPar multi-language flag, mask and field:

Syntax is "[0|1] /MASK=mask /FIELD=field"

Return codes:

0 - OK

1 - Illegal argument syntax

2 - mask not found by mas_ret_mas

3 - field not found by mas_ret_fld

LogiView Example:

RES = @mas_set_mlf_fld("1 EDB-ART-CFR TEST_FIELD_JAP)"

The C function dal_usx_mas_set_mlf_fld implements this userexit.

## New I18N related C functions

This is the list of new C functions that have been implemented to support UTF-8 encoded content in multi-language fields.

The prototypes are defined in $ep_root/axalant/inc/dal_pub.h

## Database Interface Module (dal_dbi)

### dal_dbi_get_lng_lis_enc

DtvLong dal_dbi_get_lng_lis_enc(dal_Handle hDal, char caaLngName[][21], char caaLngEnc[][21], char caLngKey[][4]);

Returns the names, shortcuts and encodings of the ten languages defined in the DataView repository.

## DataView Module (dal_Dtv)

### dal_Dtv_createEncodedString

DalEncodedString dal_Dtv_createEncodedString(dal_Handle hDal, const char* encoding, const char* text);

Creates an abstract data type that combines encoding and text.

### dal_Dtv_freeEncodedString

void dal_Dtv_freeEncodedString(dal_Handle hDal, DalEncodedString *encStr);
Frees an encoded string that has been created with dal_Dtv_createEncodedString.

### dal_Dtv_getString

const char* dal_Dtv_getString(dal_Handle hDal, DalEncodedString encStr);
Retrieves the text part of an encoded string. The retrieved pointer gets invalid as soon as encStr is destroyed.

### dal_Dtv_getEncoding

const char* dal_Dtv_getEncoding(dal_Handle hDal, DalEncodedString encStr);
Retrieves the encoding of an encoded string.

### dal_Dtv_appendString

void dal_Dtv_appendString(dal_Handle hDal, DalEncodedString target, DalEncodedString source);
Append the source string to the target string and change the encoding of the target string to UTF-8 if one of the string is UTF-8 encoded.

## Language Module (dal_lng)

### dal_lng_get_fld

char* dal_lng_get_fld(dal_Handle hDal, const char *cpTab, const char *cpFld, const char* cpLng,

DtvHandle hMas, int isSys, DtvHandle* hpFld);

Get the fully qualified field name and the field handle for a given field. The algorithm tries several field names:

- the field name with language abbreviation (if passed),
- the field as passed by the caller,
- the field with appended system or user language abbreviation,
- the field with appended default language abbreviation.

### dal_lng_ret_enc

const char* dal_lng_ret_enc(dal_Handle hDal, const char *cpLng);

Returns the encoding used by the specified language, or NULL if this language uses the standard 8-bit encoding.

## Mask Module (dal_mas)

### dal_mas_ret_enc_fld

const char *dal_mas_ret_enc_fld(dal_Handle hDal, DtvHandle hFld);

Returns the encoding used by the specified field, or NULL if this language uses the standard 8-bit encoding.

#### dal_mas_ret_siz_byt_fld

DtvLong dal_mas_ret_siz_byt_fld(dal_Handle hDal, DtvHandle hField);

Returns the maximum number of bytes needed to store the contents of this field.

#### dal_mas_ret_siz_chr_fld

DtvLong dal_mas_ret_siz_chr_fld(dal_Handle hDal, DtvHandle hField);

Returns the maximum number of charecters that can be stored in this field.

#### dal_mas_ret_chr_len_fld

DtvLong dal_mas_ret_chr_len_fld(dal_Handle hDal, DtvHandle hField);

Returns the number of bytes needed to store one character of this field.

#### dal_mas_rea_enc_fld

DtvLong dal_mas_rea_fld_enc(dal_Handle hDal, DtvHandle hFld, DtvLong lRow, char ** cppStr, const char *cpEnc);

Returns the field content converted to the specified target encoding.

## Menu Module (dal_men)

#### dal_men_rea_row_enc

DalEncodedString dal_men_rea_row_enc (dal_Handle hDal, DtvHandle idm, DtvLong row);

Read the selection text from the passed menu row.

#### dal_men_wri_row_enc

DtvLong dal_men_wri_row_enc (dal_Handle hDal, DtvHandle idm, DtvLong row, DalEncodedString str_txt, char *str_fnc, char *str_par);

Write the passed selection to the given row.

## Utility Module (dal_uti)

#### dal_uti_ret_std_enc

const char *dal_uti_ret_std_enc(dal_Handle hDal);

Returns the name of the current 8-bit encoding used by the server.

#### dal_uti_ret_utf_enc

const char *dal_uti_ret_utf_enc(dal_Handle hDal);

Returns the name of the current UTF encoding used by the server.

#### dal_uti_toUTF8

size_t dal_uti_toUTF8(dal_Handle hDal, const char *cpStr, char **cppUtf, int *ipErrorCode);

Converts a string in 8-bit standard encoding into its UTF-8 represenation.

#### dal_uti_fromUTF8

size_t dal_uti_fromUTF8(dal_Handle hDal, const char *cpUtf, char **cppStr, int *ipErrorCode);

Converts an UTF-8 string into the 8-bit standard encoding.

### Standard Trigger Module (dal_wdh)

#### dal_wdh_ret_sig_enc

char* dal_wdh_ret_sig_enc(dal_Handle hDal, DtvLong lRow, const char *cpEnc);

Returns the signature of a row converted to the specified target encoding.

## ECI

### New server side ECI parameter and connection API

To support I18N, it was necessary to modify and enhance the server side ECI structures and functions. Therefore, the ECI on server side is now different from the ECI on client side.

This allows server side modules to use both API at the same time and in the same compilation unit. For instance the EIP C library implements some server ECI functions, but it also opens an ECI connection to a remote server.

To support this, the following modifications have been implemented for Agile e6.0.2:

### New data types Eci_param and Eci_param_len

- The Eci_param structure is based on the client side eci_param structure, but it has an additional member enc to store the character encoding used by the parameter.
- The Eci_param_len data type is used within the Eci_param structure and replaces the client side data type eci_param_len.

### Parameter API

The New server module ecipar_enc implements the server side ECI parameter API that supports the character conversion needed for I18N. The module is part of the Cci library (epsrv_cci.dll/libepsrv_cci.so).

The following functions are provided:

```
int Eci_add_par ( Eci_param * eci_par, char *daten );
int Eci_add_par_fmt(Eci_param *tpPar, const char *cpFormat, ...);
int Eci_add_blb ( Eci_param * eci_par, char *cp_Blob, int i_SizeBlob);
int Eci_add_blb_fil ( Eci_param * eci_par, char *cp_BlobFile);
int Eci_add_ltx ( Eci_param * eci_par, char *cpLtxStr);
int Eci_end_par ( Eci_param * eci_par );
int Eci_get_par ( Eci_param * eci_par, char *daten );
int Eci_fet_par ( Eci_param * eci_par, char *daten, int *ipSize);
int Eci_get_blb_ptr ( Eci_param * eci_par, char **cpp_Blob, int *ip_BlobSize);
int Eci_get_blb_fil_ptr ( Eci_param * eci_par, char *cp_BlobFile, int *ip_BlobSize);
int Eci_get_ltx_ptr (Eci_param *eci_par, char **cppLtxFld, int *iSizeLtxFld);
int Eci_get_par_ptr ( Eci_param * eci_par, char **daten );
int Eci_fet_par_ptr ( Eci_param *eci_par, char **daten );
int Eci_fet_par_ptr_fre ( Eci_param *eci_par, char **daten, List_Handle hFreeList );
int Eci_lis_to_par ( Eci_param * par, const char **list );
int Eci_par_to_lis ( const char **list, Eci_param * par, List_Handle *hpFreeList );
int Eci_fre_par_lis ( List_Handle *hpFreeList );
int Eci_red_par ( Eci_param * eci_par, char *datastri, Eci_param_len nbytes );
int Eci_ret_par_num ( Eci_param * eci_par );
int Eci_set_par ( Eci_param * eci_par, int numpar );

int Eci_des_par (Eci_param * eci_par);
int Eci_cat_par(Eci_param *tpDest, Eci_param *tpSource);
int Eci_fre_par_lis ( List_Handle *hpFreeList );

int Eci_cre_par (Eci_param * eci_par, Eci_param_len nbytes);
int Eci_def_par (Eci_param * eci_par, char *datastri, Eci_param_len nbytes);


/* These functions are only available when running inside the server */
int Eci_cre_par_enc (Eci_param * eci_par, Eci_param_len nbytes, const char *enc);
int Eci_def_par_enc (Eci_param * eci_par, char *datastri, Eci_param_len nbytes, const char *enc);
```

```
int Eci_add_par_fld(Eci_param *tpPar, DtvHandle hFld, DtvLong lRow);
int Eci_get_par_fld(Eci_param *tpPar, DtvHandle hFld, DtvLong lRow);
int Eci_add_par_fld_str(Eci_param *tpPar, DtvHandle hFld, const char *cpStr);
int Eci_get_par_fld_str(Eci_param *tpPar, DtvHandle hFld, char **cppStr);
int Eci_cnv_par_to_fld(Eci_param *tpPar, DtvHandle hFld, const char *cpRaw, char **cppStr);
int Eci_get_par_fld_sel(Eci_param *tpPar, DtvHandle hFld, char **cppStr);

const char *Eci_get_enc(void *hCon);

int Eci_add_par_raw ( Eci_param * eci_par, const char *daten );
int Eci_get_par_raw ( Eci_param * eci_par, char *daten );
int Eci_fet_par_raw ( Eci_param * eci_par, char *daten, int *ipSize);
int Eci_get_ltx_ptr_raw (Eci_param *eci_par, char **cppLtxFld, int *iSizeLtxFld);
int Eci_get_par_ptr_raw ( Eci_param * eci_par, char **daten );
int Eci_fet_par_ptr_raw ( Eci_param *eci_par, char **daten );
int Eci_fet_par_ptr_fre_raw ( Eci_param *eci_par, char **daten, List_Handle hFreeList );

int Eci_lis_to_par_raw ( Eci_param * par, const char **list );
int Eci_par_to_lis_raw ( const char **list, Eci_param * par, List_Handle *hpFreeList );
```

All of these functions have the prefix Eci_ to distinguish them from their client side counterparts. The implementation of these functions takes care of converting the character data from the client side encoding as stored in the parameter to and from the server side encoding.

The functions ending with _raw do no character conversion, they contain the same logic as the client side functions and simply transfer the raw string data into the parameter structure.

The header file of this module (ecipar_enc.h) contains two switches:

- ECI_SERVER:
  If set to 0, the server side API is disabled. Default value is 1.

- ECI_MAP_NAMES:
  If set to 1, all client function and type names are mapped to their new server side counterparts.

## Connection API

Two new functions have been added to the server side connection API in the edb_eci module (part of the epsrv_shr.dll/libepsrv_shr.so):

```
int Eci_call(edb_data_header *tpHeader, const char *cpFunction,
```

```
        Eci_param *tpInPar, char **cpRetCode, Eci_param *tpRetPar);
int Eci_win(edb_data_header * connection, char *CpAction);
```

- Eci_call:
  Use this function to call a server side ECI function from a server side module. Use eci_call to call a remote ECI function.

- Eci_win:
  Dummy implementation of the eci_win function.

The header file of this module (edb_eci.h) contains two switches:

- ECI_SERVER:
  If set to 0, the server side API is disabled. Default value is 1.

- ECI_MAP_NAMES:
  If set to 1, all client function and type names are mapped to their new server side counterparts.

## New ECI server function eci_has_cap

Checks if a module specific capability is available in server.

ECI-Inputparameter consists of ...

| | | |
|---|---|---|
| cpModId | string | ID of the module to check<br>(one of the EDB_MID_... constants) |
| cpCapability | string | name of capability to check<br>(one of the EDB_CAP_... constants) |

ECI-Outputparameter consists of ...

| | | |
|---|---|---|
| cpHasCapability | string | "y": capability is available<br>"n": capability is not available |

The available capabilities will be extended in future as necessary. See edb_mid.h for available Modue ID's and capabilities.

## New ECI server function eci_use_cap

ECI-Inputparameter consists of ...

| | | |
|---|---|---|
| cpModId | string | ID of the module to check |

(one of the EDB_MID_... constants)

cpCapability string name of capability to check

cpUseIt string activation flag ("on" or "off")

ECI-Outputparameter consists of ...

cpResult string new capability state:

"on" capability is active

"off": capability is inactive

"n/a": capability is not available

## New ECI parameter helper function eci_add_par_fmt

Prototype:

```
int eci_add_par_fmt(eci_param *tpPar, const char *cpFormat, ...);
```

Description:

Adds a variable argument list to an ECI parameter.

@param tpPar ECI parameter to use

@param cpFormat printf compatible format string

@param ... arguments matching cpFormat

@return return code of eci_add_par

@see Str_vformat

@see eci_add_par

## New ECI parameter helper function eci_cat_par

Prototype:

```
int eci_cat_par(eci_param *tpDest, eci_param *tpSource);
```

Description:

Concatenates two ECI parameters.

@param tpDest destination parameter

@param tpSource source parameter

@return 0 - OK

## New ECI server function eci_get_enc

Returns the encoding used by this connection.

Several variations of the encoding name are returned:

Different standards have different names or aliases for the very same codepage.
To support different environments, eci_get_enc returns the encoding names for the following standards (resource links are included for your convenience):

- IANA: Character sets
- ICU: Character Set Mapping Tables
- Java: Character Encodings
- Microsoft Windows: Globalization Step-by-Step

A nice tool to explore all the dífferent names and aliases is the ICU Converter Explorer.

Function definition:

ECI-Inputparameter is empty

ECI-Outputparameter consists of ...

cpEnc1 stringlist name of encoding and name of standard:
e.g. '"UTF-8" "IANA"'
or '"ISO8859-15" "Java"'
...
cpEncN stringlist

## New ECI server function eci_set_enc

Sets the encoding to use for this connection.

Useful to switch to **UTF-8** encoding when writing I18N capable ECI clients. Unless a client calls this function, the 8-bit standard encoding of the C server is used.

ECI-Inputparameter consists of ...

cpEnc stringlist name of encoding and - optional - name of the standard
e.g. '"UTF-8" "Java"
or '"Cp1252", "Windows"
either "UTF-8" or the 8-bit standard encoding
used by the server, e.g. "ISO8859-15".

ECI-Outputparameter is empty

If the standard name is missing, IANA is assumed.
If the server does not support the desired encoding, an ECI_WRG_PAR error code is returned. Call eci_get_enc to get the current server side encoding.

In this release, the C server only supports the following encodings:

- ISO8859-1
- ISO8859-15
- UTF-8

# Java Daemon

## Bug fix for Call 215243

The daemon supports a new startup method named "Agile e6.0.2":

If configured this way, the daemon calls the server with a new option "-c" to specify the desired port, the session ID, a daemon callback port and (optional) the port range to use.

Example:

```
axalant ... -c 5043:1139583444897:20000:5042-5099
```

The server tries to allocate the desired port. If that fails, it tries all ports within the port range beginning with the desired port + 1. If a port has been allocated, the server informs the daemon by sending a JDP request containing the port number.

If no port could be allocated, 0 is send as port number to register the failure. The daemon will then tell the client that no more ports are available.
To ensure that the registration requests are processed as fast as possbible, the daemon needs a new port especially for this purpose. The new property "RegistrationPort" contains the number of this port.

# Java Client

## Bitmaps in list field titles

The client now supports the same "icon as title" feature for lists the way it always worked for forms.

Example:

```
#myTitleIcon:myTitleText
```

The title text is optional.

This does not work in the DataView client.

# LogiView

## New LogiView trigger lgv_usx_get_tra_lev

Added new LogiView trigger lgv_usx_get_tra_lev() that returns the current EPQ transaction level. The following LogiView code seems to be correct but has some pitfalls:

```
10  start_transaction()
20  [...call some functions that set the variable RES in error case...]
30  if ( RES <> 0 )
40    abort_transaction()
50  else
60    end_transaction()
70  endif
```

If any of the called functions aborts the data base transaction but can not set RES correctly (e.g. xstate_nxt_sta), the complete transaction is always commited.

It is now possible to handle this situation correctly:

```
10  start_transaction()
20  [...call some functions that set the variable RES in error case...]
30  if ( RES <> 0 )
40    abort_transaction()
50  else
60    RES = @lgv_usx_get_tra_lev()
70    if ( RES < 0 )
80      abort_transaction()
90    else if ( RES > 0 )
100     end_transaction()
110   else
120     put("Faulty nested transactions!")
130   endif
140 endif
```

# FELICS V3.01

The new version 3.01 of the license server FELICS has been integrated. So the new provided FELICS Agents should be installed.

# Conceptual Product Structure (CPS)

Added new functionality "Parallel Structure" for modular product component structures.

# Multi Access Rights

Added new functionality "Multi Organizational Access Rights"

New no-select menu userexit "xrole_get_cfg" to get value of configuration parameter EDB-ROLE-ACTIVE. This menu userexit can also be called from any LGV procedure.

# Call 215973

Added parameter '/NO_CONFIRMATION' to trigger xchg_opr_mov(). If the option /NO_CONFIRMATION is specified, the confirmation dialog is not displayed when related change operations are moved.

# EPQ

The database adapter is loaded dynamically, now. Therefore the legacy EPQ adapter has been split into two parts, a neutral part (epq10c) and a database specific part (currently epq10c_ora101 and epq10c_mssql90 respectively). The neutral part is linked with the application server and loads the database specific part depending on a setting in the enviroment definition file or the contents of an environment variable ("EPQ_LIBRARY_NAME"):

[Database]

Library = epq10c_ora101

or

[Database]

Library = epq10c_mssql90

If the "Library" attribute or the environment variable is not defined, epq10c_ora101 is loaded and initialized.

## Oracle

- Added nvarchar support
- Added client charset conversion

## MS-SQL

- Microsoft's ADO interface is used instead of the DB-library
- Added nvarchar support
- Added client charset conversion
- Added bind variables
- Use BigInt instead of Money for Number(12) (file server)
- Use varchar instead of text

- Use static stored procedure to execute structure explosion
- Counting is always executed as "SELECT COUNT(*) FROM (<subselect>) AS DUMMY"
- Increased BLOB limit from 32762 to 2147483647 (2^31-1)
- Added "CommandTimeout" to the environment definition file
- Implemented statements like Oracle's "SELECT ... FOR UPDATE" to support number server
- Added ability to open parallel connections
- Added ability to return INT64 values (currently not used)