CHARACTER INTERNATIONALIZATION IN DATABASES: A CASE STUDY

By Hirotaka Yoshioka and Jim Melton

ABSTRACT

Character internationalization poses difficult problems for database management systems because they must address user (stored) data, source code, and metadata. The revised (1992) standard for database language SQL is one of the first standards to address internationalization in a significant way. DEC Rdb is one of the few Digital products that has a complete internationalization (Asian) implementation that is also MIA compliant. The product is still evolving from a noninternationalized product to a fully internationalized one; this evolution has taken four years and provides an excellent example of the issues that must be resolved and the approaches to resolving them. Rdb can serve as a case study for the software engineering community on how to build internationalized products.

INTRODUCTION

Internationalization is the process of producing specifications and products that operate well in many languages and cultures.[1] Internationalization has several different aspects such as character set issues, date and time representation, and currency representation. Most of these affect many areas of information technology where the solutions are reasonably similar; for example, solutions to currency representation are equally applicable to database systems and to programming languages. Database systems, however, are affected in several unique ways, all of which deal with character sets. In this paper, we focus on the issues of character set internationalization in database management systems (DBMS) and do not address the other aspects of date and time, currency, or locales.

To better understand the problems and solutions associated with character internationalization of database systems, we present an overview of the solutions found in the standard query language (SQL) standard and report a case study of implementing those solutions in a commercial product. We first discuss the character internationalization features supported in the recently published revision of the standard for Database Language SQL (ISO/IEC 9075:1992 and ANSI X3.135-1992).[2] We then describe in some detail the application of those features in DEC Rdb, Digital's relational database product. The internationalization of DEC Rdb serves as a case study, or a model, for the internationalization of Digital's software products in general.

INTERNATIONALIZATION IN THE SQL STANDARD

Like most computer languages, SQL came into being with the minimal set of characters required by the language; vendors were free to support as many, or as few, additional characters as they perceived their markets demanded. There was little, if any, consideration given to portability beyond the English language customer base. In 1989, after work was completed on ISO 9075:1989 and ANSI X3.135-1989 (SQL-89), significant changes were proposed for the next revision of the SQL database language to address the requirement for additional character set support. (Unfortunately, this put SQL in the vanguard, and little support existed in the rest of the standards community for this effort.)

Character Set Support

SQL must address a more complex set of requirements to support character sets than other programming languages due to the inherent nature of database systems. Whereas other programming languages have to cover the character set used to encode the source program as well as the character set for data processed by the program, database systems also have to address the character set of the metadata used to describe the user data. In other words, character set information must be known within three places in a database environment.

1. The user data that is stored in the database or that is passed to the database system from the application programs.

   In SQL, data is stored in tables, which are two-dimensional representations of data. Each record of data is stored in a row of a table, and each field in a row corresponds to a column of a table. All the data in a given column of a table has the same data type and, for character data, the same character set.

2. The metadata stored in the database that is used to describe the user data and its structure.

   In SQL databases, metadata is also stored in tabular form (so that it can be retrieved using the same language that retrieves user data). The metadata contains information about the structure of the user data. For example, it specifies the names of the users' tables and columns.

3. The data management source code.

   Data management statements (for querying and updating the database) have to be represented as character strings in some character set. There are three aspects of these statements that can be independently considered. The key words of the language (like SELECT or UPDATE) can be represented in one character set -- one that contains

only the alphabetic characters and a few special (e.g., punctuation) characters; the character string literals that are used for comparison with database data or that represent data to be put into the database; and the identifiers that represent the names of database tables, columns, and so forth.

Consider the SQL statement

```
SELECT EMP_ID FROM EMPLOYEES
    WHERE DEPARTMENT = 'Purchasing'
```

In that statement, the words SELECT, FROM, and WHERE; the equals sign; and the two apostrophes are syntax elements of the SQL language itself. EMP_ID, EMPLOYEES, and DEPARTMENT are names of database objects. (EMPLOYEES is a table; the other two are columns of that table.) Finally, Purchasing is the contents of a character string literal used to compare against data stored in the DEPARTMENT column.

That seems straightforward enough, but what if the database had been designed and stored in Japan so that the names of the table and its columns were in Japanese kanji characters? Furthermore, what if the name of some specific department was actually expressed in Hebrew (because of a business relationship)? That means that our database would have to be able to handle data in Hebrew characters, metadata in Japanese characters, and source code using Latin characters!

One might reasonably ask whether this level of functionality is really required by the marketplace. The original impetus for the character internationalization of the SQL standard was provided by proposals arising from the European and Japanese standards participants. However, considerable (and enthusiastic) encouragement came from the X/Open Company, Ltd. and from the Nippon Telephone and Telegraph/Multivendor Integration Architecture (NTT/MIA) project, where this degree of mixing was a firm requirement.[3]

The situation is even more complex than this example indicates. In general, application programs must be able to access databases even though the data is in a different character encoding than the application code! Consider a database containing ASCII data and an application program written in extended binary coded decimal interchange code (EBCDIC) for an IBM system, and then extend that image to a database containing data encoded using the Japanese extended UNIX code (EUC) encoding and an application program written in ISO 2022 form. The program must still be able to access the data, yet the character representations (of the same characters) are entirely different. Although the problem is relatively straightforward to resolve for local databases (that is, databases residing on the same computer as the application),

it is extremely difficult for the most general case of heterogeneous distributed database environments.

Addressing Three Issues

To support internationalization aspects, three distinct issues have to be addressed: data representation, data comparison, and multiple character set support.

Data Representation. How is the data (including metadata and source code) actually represented? The answer to this question must address the actual repertoire of characters used. (A character repertoire is a collection of characters used or available for some particular purpose.) It must also address the form-of-use of the character strings, that is, the ways that characters are strung together into character strings; alternatives include fixed number of bits per character, like 8-bit characters, or variable number of bits per character, like ISO 2022 or ASN.1. Finally, the question must deal with the character encoding (for example, ASCII or EBCDIC). The combination of these attributes is called a character set in the SQL standard.

It is also possible for the data to be represented in different ways within the database and in the application program. A column definition that specifies a character set would look like this

```
NAME CHARACTER VARYING (6)
    CHARACTER SET IS KANJI,
```

or

```
NAME NATIONAL CHARACTER VARYING (6),
```

(which specifies the character set defined by the product to the national character set), while a statement that inserts data into that column might be

```
INSERT INTO EMPS(NAME)
    VALUES (..., _KANJI'**', ...);
```

If the name of the column were expressed in hiragana, then the user could write

```
INSERT INTO EMPS(_HIRAGANA 2+++)
    VALUES (..., _KANJI'**', ...);
```

[EDITOR'S NOTE: The two asterisks (**) above are not really part of the code but indicate the placement of two kanji characters in the code. So too, three hiragana characters representing NA MA E are in the code in place of the plus signs +++ shown here.]

Data Comparison. How is data to be compared? All character data has to be compared using a collation (the rules for comparing

character strings). Most computer systems use the binary values of each character to compare character data 1 byte at a time. This method, which uses common character sets like ASCII or EBCDIC, generally does not provide meaningful results even in English. It provides far less meaningful results for languages like French, Danish, or Thai.

Instead, rules have to be developed for language-specific collations, and these rules have to resolve the problems of mixing character sets and collations within SQL expressions.

Applications can choose to force a specific collation to be used for comparisons if the default collation is inappropriate:

```
WHERE :hostvar = NAME COLLATE JAPANESE
```

Multiple Character Set Support. How is the use of multiple character sets handled? The most powerful aspect of SQL is its ability to combine data from multiple tables in a single expression. What if the data in those tables is represented in different character sets? Rules have to be devised to specify the results for combining such tables with the relational join or union operations.

What if the character sets of data in the source program are different from those in the database? Rules must exist to provide the ability for programs to query and modify databases with different character sets.

Components of Character Internationalization

SQL recognizes four components of character internationalization: character sets, collations, translations, and conversions. Character sets are described above; they comprise a character repertoire, a form-of-use, and an encoding of the characters. Collations are also described above; they specify the rules for comparing character strings expressed in a given character repertoire.

Translations provide a way to translate character strings from one character repertoire to a different (or potentially the same) repertoire. For example, one could define a translation to convert the alphabetic letters in a character string to all uppercase letters; a different translation might transliterate Japanese hiragana characters to Latin characters. By comparison, conversions allow one to convert a character string in one form-of-use (say, two octets per character) into another (for example, compound text, a form-of-use defined in the X Window System).

SQL provides ways for users to specify character sets,

collations, and translations based on standards and on vendor-provided facilities. The current draft of the next version of the SQL standard (SQL3) also allows users to define their own character sets, collations, and translations using syntax provided in the standard.[4,5] If these facilities come to exist in other places, however, they will be removed from the SQL standard (see below). SQL does not provide any way for users to specify their own conversions; only vendor-provided conversions can be used.

Interfacing with Application Programs

Application programs are typically written in a third-generation language (3GL) such as Fortran, COBOL, or C, with SQL statements either embedded in the application code or invoked in SQL-only procedures by means of CALL-type statements.[6] As a result, the interface between the database system and 3GL programs presents an especially difficult problem in SQL's internationalization facilities. Figure 1 illustrates the procedure to invoke SQL from C; Figure 2 shows SQL as it is invoked from C; and Figure 3 shows SQL schema.

Figure 1 Invoking SQL from C

```
main()
{
  #include <stdio.h>
  #include <stdlib.h>
  #include "SQL92.h"  /* Interface to SQL-92 */

  static sqlstate char[6];
  static employee_number char[7];
  static employee_name wchar_t[26];
  static employee_contact char[13];

  /* Assume some code here to produce an appropriate
     employee number value */

  LOCATE_CONTACT (employee_number, employee_name,
                  employee_contact, sqlstate);

  /* Assume more code here to use the result */

}
;
```

Figure 2 SQL Invoked from C

```
MODULE i18n_demo NAMES ARE Latin1
  LANGUAGE C
```

```
    SCHEMA personnel AUTHORIZATION management

    PROCEDURE locate_contact
        ( :emp_num           CHARACTER (6) CHARACTER SET Ascii,
          :emp_name          CHARACTER VARYING (25) CHARACTER SET Unicode,
          :contact_name      CHARACTER VARYING (6) CHARACTER SET Shift_jis,
          SQLSTATE)
      SELECT name, contact_in_japan
        INTO :emp_name, :contact_name
        FROM personnel.employees
        WHERE emp_id = :emp_num;
```

Figure 3 SQL Schema

```
  CREATE SCHEMA personnel AUTHORIZATION management
    DEFAULT CHARACTER SET Unicode


  CREATE TABLE employees (
      emp_id                 CHARACTER (6) CHARACTER SET Ascii,
      name                   CHARACTER VARYING (25),
      department             CHARACTER (10) CHARACTER SET Latin1,
      salary                 DECIMAL (8,2),
      contact_in_japan       CHARACTER VARYING (6) CHARACTER SET Shift_jis,
      ...,
      PRIMARY KEY (emp_id) )
```

In these figures, all the metadata values (that is, the identifiers) are expressed in Latin characters; this resolves the data representation issue. The reader should compare the character sets of the data items in the EMPLOYEES table and the corresponding parameters in the SQL procedure. The difficulties arise when trying to achieve a correlation between the parameters of the SQL procedure and the arguments in the C statement that invokes that procedure.

The C variable employee_number corresponds to the SQL parameter :emp_num; the C data type char is a good match for CHARACTER SET ASCII. The C variable employee name corresponds to the SQL parameter :emp_name; the C data type wchar_t is chosen by many vendors to match CHARACTER SET Unicode. However, CHARACTER SET Shift_jis is more complicated; there is no way to know exactly how many bytes the character string will occupy because each character can be 1 or 2 bytes in length. Therefore, we have allocated a C char that permits up to 13 bytes. Of course, the C run-time library would have to include support for ASCII data, Unicode data, and Shift JIS data.

Typically, 3GL languages have little or no support for character sets beyond their defaults. Consequently, when transferring data from an internationalized SQL database into a noninternationalized application program, many of the benefits

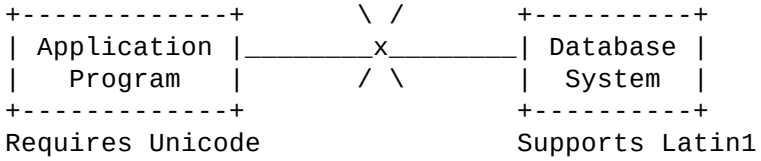
are lost. Happily, that situation is changing rapidly. Programming language C is adding facilities for handling additional character sets, and the ISO standards group responsible for programming languages (ISO/IEC JTC1/SC22) is investigating how to add those capabilities to other languages as well.

The most difficult issue to resolve concerns the differences in specific character sets (especially form-of-use) supported by SQL implementations and 3GL implementations. As with other issues, purely local situations are easy to resolve because a DBMS and a compiler provided by the same vendor are likely to be compatible. Distributed environments, especially multivendor ones, are more complicated. SQL has provided one solution: it permits the user to write SQL code that translates and converts the data into the form required by the application program as long as the appropriate conversions and translations are available for use by SQL. Of course, once the data has been transferred into the application program, the question remains: What facilities does it have to manipulate that data?

Remote Database Access Issue

As mentioned, a distributed environment presents significant difficulties for database internationalization. A simple remote database access scenario illustrates these problems. If an application program must access some (arbitrary) database via a remote (e.g., network) connection, then the remote database access facility must be able to deal with all the character sets that the application and database use together; it may also have to deal with differences in available character sets. (See Figure 4.)

Figure 4 Remote Database Access



An ISO standard for remote database access (ISO/IEC 9579-1 and 9579-2) uses the ASN.1 notation and encoding for transporting SQL commands and database data across remote connections.[7] ASN.1 notation, as presently standardized, provides no way to use various character sets in general. Recently work has begun to resolve this problem. The revised standard must allow a character set to be specified uniquely by means of a name or identifier that both ends of the connection can unambiguously interpret in

the same way. The individual characters in ASN.1 character strings must be similarly identifiable in a unique way.

This problem has not yet been resolved in the standards community, partly because several groups have to coordinate their efforts and produce compatible solutions.

Hope for the Future

In the past, programming languages, database systems, networks, and other components of information management environments had to deal with character sets in very awkward ways or use vendor-provided defaults. The result has been an incredible mess of 7-bit (ASCII, for example) and 8-bit (Latin-1, for example) code sets, PC code pages, and even national variants to all of these. The number of code variants has made it very difficult for a database user to write an application that can be executed on any database system using recompilation only. Collectively, they make too many assumptions about the character set of all character data.

The future outlook for database internationalization was improved dramatically by the recent adoption of ISO 10646, Universal Multiple-Octet Coded Character Set (UCS) and an industry counterpart, Unicode.[8] The hope is that Unicode will serve as a "16-bit ASCII" for the future and that all new systems will be built to use it as the default character set.

Of course, it will be years -- if not decades -- before all installed computer hardware and software use Unicode. Consequently, provisions have to be made to support existing character sets (as SQL-92 has done) and the eccentricities of existing hardware and software (like networks and file systems). As a result, several different representations of Unicode have been developed that permit transmission of its 16-bit characters across networks that are intolerant of the high-order bit of bytes (the eighth bit) and that permit Unicode data to be stored in file systems that deal poorly with all the bit patterns it permits (such as octets with the value zero).

In the past few years, many alternative character representations have been considered, proposed, and implemented. For example, ISO 2022 specifies how various character sets can be combined in character strings with escape sequences and gives instructions on switching between them.[9] Similarly, ASN.1-like structures, which provide fully tagged text, have been used by some vendors and in some standards, e.g., Open Document Architecture.[10] None of these representations has gained total acceptance. Database implementors perceive difficulties with a stateful model and with the potential performance impact of having a varying number of bits or octets for each character. UCS and Unicode appear to be likely to gain wide acceptance in the database arena and in other areas.

Future Work for the SQL Standard

One should not conclude that the job is done, that there is nothing left to work on. Instead, a great deal of work remains before the task of providing full character set internationalization for database systems is completed.

At present, the working draft for SQL3 contains syntax that would allow users to define their own character sets, collations, and translations using a nonprocedural language.[4,5] In general, the SQL standards groups believe that it is inappropriate for a database standard to specify language for such widely needed facilities. Consequently, as soon as the other responsible standards bodies provide a language for these specifications, it is probable that this capability will be withdrawn from the SQL3 specification. This decision would completely align the SQL character internationalization capabilities with the rest of the international standards efforts.

After other standards for these tasks are in place, however, the remote data access (RDA) standard will have to be evolved to take advantage of them. RDA must be able to negotiate the use of character sets for database applications and to transport the information between database clients and servers. In order for RDA to be able to do this, the ASN.1 standard will have to support arbitrary named character sets and characters from those sets.

As a result, relevant standards bodies will need to provide (1) names for all standardized character sets and (2) the ability for vendors to register their own character sets in a way that allows them to be uniquely referenced where needed. Still other bodies will need to provide language and services for defining collations and translations. Finally, registries will need to be established for vendor-supplied collations, translations, and conversions.

Of course, the greatest task will be to provide complete support for all these facilities throughout the information processing environment: operating systems, communication links, CPUs, printers, keyboards, windowing systems, file systems, and so forth. Healthy starts have been made on some of these (such as the X Window System), but much work remains to be done.

DEC Rdb: AN INTERNATIONALIZATION CASE STUDY

DEC Rdb (Rdb/VMS) is one of the few Digital products that has an internationalized implementation that is also compliant with the multivendor integration architecture (MIA).[11,12] Its evolution from a noninternationalized product to a fully internationalized one has taken four years to achieve. The design and development of Rdb can serve as a case study for software engineers on how to

build internationalized products. In this half of our paper, we present the history of the reengineering process. Then we describe some difficulties with the reengineering process and our work to overcome them. Finally, we evaluate the result.

Localization and Reengineering

The localization process comprises all activities required to create a product variant of an application that is suitable for use by some set of users with similar preferences on a particular platform. Reengineering is the process of developing the set of source code changes and new components required to perform localization. DEC Rdb had to be reengineered to support several capabilities that are mandatory in Japan and other Asian countries.

Our experience has shown that the reengineering process is very expensive and should be avoided. If the original product was not designed for internationalization or localization, however, reengineering is a necessary (and unavoidable) evil. Typically, reengineering is required; so we decided to develop a technology that would avoid reengineering and to build a truly internationalized product.

Most engineering groups follow the old assumptions about product design. These assumptions include the following:

1. The character set is implicitly ASCII.

2. Each character is encoded in 7 bits.

3. The character count equals the byte count and equals the display width in columns.

4. The maximum number of distinct characters is 128.

5. The collating sequence is ASCII binary order.

6. The messages are in English.

7. The character set of the source code is the same as it is at run time.

8. The file code (the code on the disk) is the same as the process code (the code in memory).

Different user environments require different product capabilities. Japanese kanji characters are encoded using 2 bytes per character. If a product assumes that the character set is 7-bit ASCII, that product must be reengineered before it can be used in Japan. On the other hand, internationalized products can operate in different environments because they provide the capabilities to meet global requirements. These capabilities include the following:

1. Multiple character sets ensure that the customer's needs are met.

2. Each character is encoded using at least 8 bits.

3. The character count does not equal the byte count or the display width.

4. The maximum number of unique characters is unknown.

5. The collating sequence meets the customer's needs.

6. The messages are in the language the customer uses.

7. The character set of the source code is not necessarily the same as it is at run time.

8. The file code is not necessarily the same as the process code.

The reengineering process has two significant drawbacks: (1) the high cost of reengineering and (2) the time lag between shipping the product to the customer in the United States and shipping to the customer in Japan. The time lag can be reduced but cannot be eliminated as long as we reengineer the original product. If a local product is released simultaneously with the original, both Digital and the customers will benefit significantly.

In the next section, we follow the DEC Rdb product through the reengineering process required to produce the Japanese Rdb version 3.0.

REENGINEERING PROCESS

DEC Rdb version 3.0 was a major release and consequently was very important to the Japanese market. The International System Engineering Group was asked to release the Japanese version by the end of 1988, which was within six months of the date that it was first shipped to customers in the United States.

Japanese and Asian Language Requirements to VAX Rdb/VMS

Japanese and Asian language requirements apply to DEC Rdb and other products as well. The requirements common to Asian languages are 2-byte character handling, local language editor support, and message and help file translation.

Japanese script uses a 2-byte code, therefore 2-byte character handling is mandatory. For example, character searches must be performed on 2-byte boundaries and not on 1-byte boundaries. If a string has the hexadecimal value 'A1A2A3A4', then its substrings are 'A1A2' and 'A3A4'. 'A2A3' must not be matched in the string.

Digital's Asian text editors, e.g., the Japanese text processing utility (JTPU) and Hanzi TPU (for China), must be supported as well as the original TPU, the standard EDT editor, and the language-sensitive editor.

Messages, help files, and documentation must all be translated into local languages.

The country-specific requirements include support for a Japanese input method. Kana-to-kanji input methods must be supported in command lines. In addition, 4-byte character handling is required for Taiwan (Hanyu). Finally, NTT/MIA SQL features must be added for Japan.

Since there are not many requirements, one might conclude that the reengineering task is not difficult. However, reengineering is complicated, expensive, and time consuming; and thus should be avoided.

Reengineering Japanese Rdb Version 3.x

A database management system like DEC Rdb is very complex. The source code is more than 810,000 lines; the build procedures are complicated; and a mere subset of the test systems consumes more than one gigabyte of disk space. Consequently, the reengineering process is complicated. The process encompasses more than modifying the source code. Instead, a number of distinct steps must be accomplished:

1. Source code acquisition
2. Build environment acquisition
3. Test system acquisition
4. Creation of the development environment for the Japanese version
5. Study of the original code
6. Modification of the source code
7. Test of the results, including the new Japanese functionality and a regression test of the original functionality
8. Maintenance of the reengineered system

Figure 5 shows the development cost in person-weeks for each of the eight steps. Two engineers stabilized the development

environment -- compile, link/build, and run -- for version 3.0 of DEC Rdb in approximately four months. It is likely that the process required four months because it was our first development work on DEC Rdb. In addition, approximately two months were needed to be able to run the test system. It was not an easy task.

[Figure 5: (Reengineering Process for Japanese Rdb Version 3.x) is not available in ASCII format.]

Each step had to be repeated for each version of the original. (Project time decreased a little.) Every version required this reengineering, even if no new functionality was introduced. The cost of building the environment became cheaper after the first time. The other steps such as modifying the source code, testing, and maintenance remained at almost the same cost.

Reengineering Metric

We modified about 10 percent of the original source modules during reengineering. Most of the modification occurred in the front end, e.g., SQL and RDML (relational database manipulation language). The engine parts, the relational database management system (RDMS), and KODA (the kernel of the data access, the lowest layer of the physical data access) were not modified very much. Table 1 gives the complete reengineering metrics.

$$\text{Reengineering metric} = \frac{(\text{modified modules} + \text{new created modules})}{(\text{original} + \text{modified} + \text{new created modules})}$$

Table 1 Reengineering Metrics

Facility	Reengineering Metric	Modified Modules	Total Modules	Size in Kilo Lines
SQL	6.3%	8	128	226.0
RDML	11.7%	11	94	188.3
RDMS	3.1%	4	127	154.0
KODA	0.6%	1	157	109.8
RMU	0.0%	0	41	80.5
Dispatcher	0.0%	0	30	60.9

Notes:

RMU is the Rdb management utility; it is used to monitor, back up, restore, and display DEC Rdb databases.

The reengineering metric for JCOBOL (a Digital COBOL compiler sold in Japan) is 47/258 = 18.2%; the size is 225.0 kilo lines.

COENGINEERING PROCESS: NO MORE REENGINEERING

To reduce and eliminate reengineering, we have taken a conservative, evolutionary approach rather than a revolutionary one. We used only proven technologies. The evolution can be divided into three phases:

1. Joint Development with Hong Kong. Our development goal was to merge Japanese, Chinese (People's Republic of China and Taiwan), and Korean versions into one common Asian Rdb source code.

2. Coengineering Phase I. Our goal was to merge Asian common Rdb into the original master sources for version 4.0. The merger of J-Rdb and Chinese-Rdb into Rdb would eliminate reengineering and create one common executable image.

3. Coengineering Phase II. In the final phase, our goal was to develop the internationalized product for version 4.2 by adding more internationalization functionality, SQL-92 support, MIA support for one common executable, and multiple character set support.

Coengineering is a development process in which local engineers temporarily relocate to the Central Engineering Group in the United States to develop the original product jointly with Central Engineering. The engineers from a non-English-speaking country provide the user requirements and the cultural-dependent technology (e.g., 2-byte processing and input methods), and Central Engineering provides the detailed knowledge of the product. This process promotes good experiences for both parties. For example, the local engineers learn the corporate process, and the corporate engineers have more dedicated time to understand the requirements and difficulties of local product needs, what internationalization means, and how to build the internationalized product. Coengineering minimizes the risks associated with building internationalized products.

Asian Joint Development

Our goal for the Asian joint development process was to provide a common Asian source code for Japan, People's Republic of China

(PRC), Taiwan, and Korea. One common source code would reduce reengineering costs in Asia. To achieve our goal, we devised several source code conventions. The purposes of the conventions were

1. To identify the module for each Asian version by its file name

2. To make it possible to create any one of the Asian versions (for Japan, the PRC, Taiwan, or Korea) or the English version from the common source codes, using conditional compilation methods

3. To identify the portions of codes that were modified for the Japanese version

4. To facilitate an engineer in Hong Kong who is developing versions for the PRC, Taiwan, and Korea

We developed the Japanese Rdb version 3.0 in Japan. The files were transferred to Hong Kong to develop versions for the PRC, Taiwan, and Korea. The modified versions were sent back to Japan to be merged into one common Asian source file.

Since we had one common Asian source file, reengineering in Hong Kong was reduced. Reengineering in Japan, however, was still necessary. We used compilation flags to create four country versions, that is, we had four sets of executable images. As a result, we needed to maintain four sets of development environments (source codes, tests, and so forth). We wanted to further simplify the process and therefore entered the coengineering phases.

Coengineering Phase I

The integration of Asian DEC Rdb into the base DEC Rdb product took place in two phases. In the first phase, we integrated the Asian code modifications into the source modules of the base product. Consequently, the specific Asian versions of the product can be attained by definition and then translation of a logical name (a sort of environment variable). No conditional compilation is necessary. In all releases of DEC Rdb version 3.x, source modules of the base product were conditionally compiled for each Asian version, which created separate object files and images.

The process steps in this phase were

1. Merge the source code

    a. Create one executable image

    b. Remove Japanese/Asian VMS dependency

c. Remove kana-to-kanji input method

2. Transfer the J-Rdb/C-Rdb tests

Source Code Merge (Rdb Version 4.0). To create a single set of images, we removed the compilation flags and introduced a new way of using the Asian-specific source code. We chose to do this by using a run-time logical name; the behavior of DEC Rdb changes based on the translation of that logical name.

We removed the Japanese/Asian VMS dependencies by using Rdb code instead of JSYSHR calls. (JSYSHR is the name given to the OpenVMS system services in Japanese VMS.)

We removed the kana-to-kanji input method: By calling LIB$FIND_IMAGE_SYMBOL (an OpenVMS system service to dynamically link library routines) to invoke an input method, the image need not be linked with JVMS; even an end user can replace an input method.

Run-time Checking. We removed the compilation flags, but introduced a new logical name, the RDB$CHARACTER_SET logical, to switch the behavior of the product. For example, if RDB$CHARACTER_SET translates to DEC_KANJI, then the symbol ARDB_JAPAN_VARIANT is set true. This would indicate that all text would be treated as if it were encoded in DEC_KANJI. The code would behave as if it were DEC J-Rdb. This translation must occur at all levels of the code, including the user interface, DEC Rdb Executive, and KODA.

Since DEC Rdb checks the value of the logical name at run time, we do not need the compilation flags; that is, we can have one set of executable images.

Figure 6 shows the values that are valid for the RDB$CHARACTER_SET logical.

Figure 6 RDB$CHARACTER_SET Logical

```
$ DEFINE RDB$CHARACTER_SET -
   { DEC_KANJI | DEC_HANZI | DEC_HANGUL | DEC_HANYU }

  DEC_KANJI         Japanese
  DEC_HANZI         Chinese
  DEC_HANGUL        Korean
  DEC_HANYU         Taiwan

$ SET LANGUAGE JAPANESE ! If you use Japanese VMS
```

The DEC J-Rdb source contains code fragments similar to those shown in Figure 7, which were taken from RDOEDIT.B32 (written in the BLISS programming language). This code was changed to use a

run-time flag set as a result of translation of the logical RDB$CHARACTER_SET, as shown in Figure 8.

Figure 7 Compilation Flag in DEC Rdb Version 3

```
! This example switches the default TPU shareable
! image (TPUSHR). If the Japanese variant is set,
! then the default editor should be JTPUSHR.
!
%IF $ARDB_JAPAN_VARIANT
%THEN
        TPU_IMAGE_NAME = ( IF (.TPU_NAME EQL 0)
        THEN $DESCRIPTOR ('TPUSHR')
        ELSE $DESCRIPTOR ('JTPUSHR'));
%ELSE
        TPU_IMAGE_NAME = $DESCRIPTOR ('TPUSHR');
```

Figure 8 Run-time Checking in Version 4

```
 ! This code could be translated to the following
 ! which might contain redundant code but should work:
 !
IF.ARDB_JAPAN_VARIANT ! If ARDB_JAPAN_VARIANT flag is true,
 THEN ! then Rdb/VMS should use the J-Rdb/VMS behavior.
      TPU_IMAGE_NAME = ( IF (.TPU_NAME EQL 0)
      THEN $DESCRIPTOR ('TPUSHR')
      ELSE $DESCRIPTOR ('JTPUSHR'))
ELSE
      TPU_IMAGE_NAME = $DESCRIPTOR ('TPUSHR');
```

Remove Japanese VMS (JVMS) Dependency. The Japanese version of DEC Rdb version 3.x used the JVMS run-time library (JSY routines). The JSY routines are Japanese-specific character-handling routines such as "get one kanji character" and "read one kanji character." The library is available only on JVMS; native VMS does not have it, so DEC Rdb cannot use it. To remove the JVMS dependency, we modified all routines that called JSY routines so that they contain their own code to implement the same functions.

The J-Rdb/VMS source contains code fragments similar to the ones shown in Figure 9. The code was changed to remove references to the JSY routines as shown in Figure 10. This example does not use JSY routines like JSY$CH_SIZE or JSY$CH_RCHAR.

Figure 9 Using JSY Routines in DEC Rdb Version 3

```
%IF $ARDB_COMMON_VARIANT %THEN
!+
! ARDB: Advance character pointer.
!
```

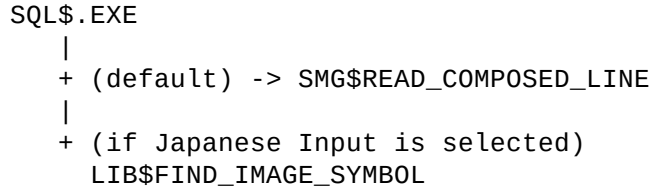
```
        ! JSY$CH_SIZE counts the size of the character.
        ! If it is ASCII, return 1,
        ! If it is Kanji, return 2.
        ! CP is a character pointer
        CP = CH$PLUS( .CP, JSY$CH_SIZE( JSY$CH_RCHAR( .CP ) ) );
        !-
        %ELSE
        CP = CH$PLUS( .CP, 1 );
        %FI !$ARDB_COMMON_VARIANT
```

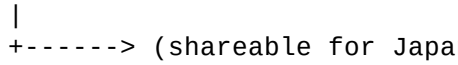
Figure 10 Removing JSY Routines in Version 4

```
    !******************run time checking

    IF $RDMS$ARDB_COMMON THEN
    !+
    ! ARDB: Advance character pointer.
    !
    ! If the code value of CP is greater than 128,
    ! then it means the first byte of Kanji, so
    ! advance 2, else it is ASCII, advance 1.
    !
    CP = CH$PLUS( .CP, (IF CH$RCHAR( .CP) GEQ 128
                        THEN
                            2
                        ELSE
                            1) );
    !-
    ELSE
    CP = CH$PLUS( .CP, 1 );
    FI !$RDMS$ARDB_COMMON

    where $RDMS$ARDB_COMMON is a macro.
```

Remove Kana-to-kanji Input Method. The dependency on JVMS can be eliminated by making the 2-byte text handling independent of JSY routines, but the input method still depends on JSYSHR for kana-to-kanji conversions. To remove this dependency, we developed a new method to invoke the kana-to-kanji conversion routine. Figure 11 shows the new input method.

Figure 11 Input Method for Version 4: Kana-to-kanji Conversion (Japanese Input) Shareable Image





Since LIB$FIND_IMAGE_SYMBOL is used to find the Japanese input at run time, JSYSHR does not need to be referenced by the SQL$ executable image.

We created a shareable image for the input method, using the SYS$LANGUAGE logical to switch to the Japanese input method or to other Asian language input methods. Since an input method is a shareable image, a user can switch input methods by redefining the logical name to identify the appropriate image.

Note that the input method is a mechanism to convert alphabetic characters to kanji characters. It is necessary to permit input of ideographic characters, i.e., kanji, through the keyboard. Asian local language groups would be responsible for creating a similar shareable image for their specific input methods.

Transfer DEC J-Rdb and DEC C-Rdb Tests. To ensure the functionality of Japanese/Asian DEC Rdb, we transferred the tests into the original development environment. We integrated not only the source modules but also all the tests. Consequently, the Asian 2-byte processing capabilities have now been tested in the United States.

Kit Components and J-Rdb Installation Procedure. The original DEC Rdb version 4.0 has the basic capability to perform 2-byte processing. Japanese and other Asian language components must be provided for local country variants. The localization kit for Japan contains Japanese documentation such as messages and help files, an input method, and the J-Rdb license management facility (LMF). As a result, we need not reengineer the original product any more. The installation procedure is also simplified. Users worldwide merely install DEC Rdb and then install a localization kit if it is needed.

The localization kits contain only the user interfaces, so no reengineering is necessary; however, translation of documentation, message files, help files, and so on to local languages still remains necessary. Nonetheless, the reengineering process is eliminated.

In version 4.0, we achieved the main goal, to integrate the Asian source code into the base product to avoid reengineering. The Japanese localization kit was released with a delay of about one month after the U.S. version (versus a five-month delay in version 3.0). The one-month delay between releases is among the best in the world for such a complex product.

Coengineering Phase II

In the second phase of integration, we redesigned the work done in Phase I and developed a multilingual version of Rdb/VMS.

In version 4.0, we introduced the logical name RDB$CHARACTER_SET to integrate Asian functionality into DEC Rdb. In Phase II, we created an internationalized version of DEC Rdb. We retained the one set of images and introduced new syntax and semantics. We also provided support for the NTT/MIA requirements.

The following are the highlights of the release. The details are given in the Appendix.

- NTT/MIA SQL Requirements
  - NATIONAL CHARACTER data type
  - N'national' literal
  - Kanji object names
- Changes/extensions to the original DEC Rdb
  - Add a character set attribute
  - Multiple character set support
- Dependencies upon other products
  - CDD/Plus, CDD/Repository: Add a character set attribute
  - Programming languages: COBOL, PIC, N

Since we are no longer reengineering the original product, we now have time to develop the new functionality that is required by NTT/MIA. The new syntax and semantics of the character-set handling are conformant with the new SQL-92 standard. As far as we know, no competitor has this level of functionality.

If we had to continue to reengineer the original, we would not have had enough resources to continue development of important new functionalities. Coengineering not only reduces development cost but also improves competitiveness.

We introduced the RDB$CHARACTER_SET logical during Phase I to switch the character set being used. Since the granularity of character set support is on a process basis, however, a user cannot mix different character sets in a given process. In Phase II, we implemented the CHARACTER SET clause, defined in SQL-92, to allow multiple character sets in a table.

Database Character Sets. The database character sets are the character sets specified for the attached database. Database character set attributes are default, identifier, and national.

SQL uses the database default character set for two elements: (1) database columns with a character data type (CHARACTER and CHARACTER VARYING) that do not explicitly specify a character set and (2) parameters that are not qualified by a character set. The user can specify the database default character set by using the DEFAULT CHARACTER SET clause for CREATE DATABASE.

SQL uses the identifier character set for database object names such as table names and column names. The user can specify the identifier character set for a database by using the IDENTIFIER CHARACTER SET clause for CREATE DATABASE.

SQL uses the national character set for the following elements.

- For all columns and domains with the data type NATIONAL CHARACTER or NATIONAL CHARACTER VARYING and for the NATIONAL CHARACTER data type in a CAST function
- In SQL module language, all parameters with the data type NATIONAL CHARACTER or NATIONAL CHARACTER VARYING
- For all character-string literals qualified by the national character set, that is, the literal is preceded by the letter N and a single quote (N')

The user can specify the national character set for a database by using the NATIONAL CHARACTER SET clause for CREATE DATABASE.

The following example shows the DEFAULT CHARACTER SET, IDENTIFIER CHARACTER SET, and NATIONAL CHARACTER SET clauses for CREATE DATABASE.

```
CREATE DATABASE FILENAME ENVIRONMENT
        DEFAULT CHARACTER SET DEC_KANJI
        NATIONAL CHARACTER SET KANJI
        IDENTIFIER CHARACTER SET DEC_KANJI;

CREATE DOMAIN DEC_KANJI_DOM CHAR(8);
CREATE DOMAIN KANJI_DOM NCHAR(6);
```

DEC_KANJI_DOM is a text data type with DEC_KANJI character set, and KANJI_DOM is a text data type with KANJI character set. The database default character set is DEC_KANJI and the national character set is KANJI.

As previously stated, the user can choose the default and identifier character sets of a database. Consequently, users can have both text columns that have character sets other than 7-bit ASCII and national character object names (i.e., kanji names, Chinese names, and so on).

In Rdb version 3.1 and prior versions, the character set was ASCII and could not be changed. In Rdb version 4.0, users could change character sets by defining the RDB$CHARACTER_SET logical. It is important to note that the logical name is a volatile attribute; that is, the user must remember the character set being used in the database in his process. On the other hand, the database character sets introduced in version 4.2 are persistent attributes, so the user is less likely to become confused about the character set in use.

Session Character Sets. The session character sets are used during a session or during the execution of procedures in a module. The session character set has four attributes: literal, default, identifier, and national.

SQL uses the literal character set for unqualified character string literals. Users can specify the literal character set only for a session or a module by using the SET LITERAL CHARACTER SET statement or the LITERAL CHARACTER SET clause of the SQL module header, DECLARE MODULE statement, or DECLARE ALIAS statement.

Session character sets are bound to modules or an interactive SQL session, and database character sets are attributes of a database. For example, a user can change the session character sets for each SQL session; therefore, the user can attach to a database that has DEC_MCS names and then attach to a new database that has DEC_HANZI names.

Octet Length and Character Length. In DEC Rdb version 4.1 and prior versions, all string lengths were specified in octets. In other words, the numeric values specified for the character-column length or the start-off set and substring length within a substring expression were considered to be octet lengths or offsets.

DEC Rdb version 4.2 supports character sets of mixed-octet and fixed-octet form-of-use. For this reason and to allow an upgrade path to SQL-92 (where lengths and offsets are specified in characters rather than octets), users are allowed to specify lengths and offsets in terms of characters. To change the default string-length unit from octet to characters, users may invoke the following:

```
SET CHARACTER LENGTH 'CHARACTERS';
```

Multiple Character Sets Examples. Users can create a domain using a character set other than the database default or national character sets with the following sequence:

```
CREATE DOMAIN DEC_KOREA_DOM CHAR(6)
    CHARACTER SET DEC_KOREAN;
```

```
CREATE TABLE TREES
        (TREE_CODE TREE_CODE_DOM,
         QUANTITY INTEGER,
         JAPANESE_NAME CHAR(30),
         FRENCH_NAME CHAR(30)
            CHARACTER SET DEC_MCS,
         ENGLISH_NAME CHAR(30)
            CHARACTER SET DEC_MCS,
         KOREAN_NAME CHAR(30)
            CHARACTER SET DEC_KOREAN,
         KANJI_NAME NCHAR(30));
```

The table TREES has multiple character sets. This example assumes the default character set is DEC_KANJI and the national character set is KANJI. Users can have object names other than ASCII names specifying the identifier character set. The database engine uses the specific routines to compare data, since the engine knows the character set of the data. With DEC Rdb version 4.2, all three issues of data representation, multiple character-set support, and data comparison have been resolved.

CONCLUSIONS

By replacing reengineering with coengineering, we reduced the time lag between shipping DEC Rdb to customers in the United States and in Japan from five months for version 3.0 in July 1988 to two weeks for version 4.2 in February 1993. Figure 12 shows the decrease in time lag for each version we developed. We also eliminated expensive reengineering and maintenance costs. Finally, we increased competitiveness.

[Figure 12: (Time Lag between U.S. and Japanese Shipment of DEC Rdb) is not available in ASCII format.]

It has taken more than four years to evolve from a noninternationalized product to an internationalized one. If the product had originally been designed to be internationalized, this process would have been unnecessary. When DEC Rdb was originally created, however, we did not have an internationalization model, the architecture, or mature techniques. Reengineering is unavoidable under these circumstances.

By sharing our experience, we can help other product engineering groups avoid the reengineering process.

FUTURE WORK FOR DEC Rdb

Coengineering has proved that an evolutionary approach is not only possible, but that it is the most reasonable approach. additional work, however, remains to be done for DEC Rdb.

DEC Rdb must support more character sets like ISO 10646-1. We think that the support of new character sets would be straightforward in the DEC Rdb implementation. DEC Rdb has the infrastructure for supporting it. SQL-92 has the syntax for it, that is, the character set clause. Furthermore, the DEC Rdb implementation has the attribute for a character set in the system tables.

Collations on Han characters should be extended. The current implementation of a collation on Han characters is based on its character value, that is, its code value. We believe the user would also like to have collations based on dictionaries, radicals, and pronunciations.[13]

SUMMARY

There are significant difficulties in the specification of character internationalization for database systems, but the SQL-92 standard provides a sound foundation for the internationalization of products. The application of SQL-92 facilities to DEC Rdb is quite successful and can serve as a case study for the internationalization for other software products.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the help and contributions made by many people during the development of DEC Rdb's internationalization facilities and those of the SQL standard. In particular, Don Blair, Yasuhiro Matsuda, Scott Matsumoto, Jim Murray, Kaz Ooiso, Lisa Maatta Smith, and Ian Smith were particularly helpful during the DEC Rdb work. During the internationalization of SQL, Laurent Barnier, David Birdsall, Phil Shaw, Kohji Shibano, and Mani Subramanyam all made significant contributions.

REFERENCES

1. G. Winters, "International Distributed Systems -- Architectural and Practical Issues," Digital Technical Journal, vol. 5, no. 3 (Summer 1993): 53-62.

2. American National Standard for Information Systems -- Database Language SQL, ANSI X3.135-1992 (American National Standards Institute, 1992). Also published as Information Technology -- Database Languages -- SQL, ISO/IEC 9075:1992 (Geneva: International Organization for Standardization, 1992).

3. W. Rannenberg and J. Bettels, "The X/Open Internationalization Model," Digital Technical Journal, vol. 5, no. 3 (Summer 1993): 32-42.

4. Database Language SQL (SQL3), Working Draft, ANSI

X3H2-93-091 (American National Standards Institute, February 1993).

5. Database Language SQL (SQL3), Working Draft, ISO/IEC JTC1/SC21 N6931 (Geneva: International Organization for Standardization, July 1992).

6. J. Melton and A. Simon, Understanding the New SQL: A Complete Guide (San Mateo, CA: Morgan Kaufmann Publishers, 1992).

7. Information Technology -- Remote Database Access -- Part 1: Generic Model, Service, and Protocol, ISO/IEC 9579-1:1993, and Information Technology -- Remote Database Access -- Part 2: SQL Specialization, ISO/IEC 9579-2:1993 (Geneva: International Organization for Standardization, 1993).

8. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," Digital Technical Journal, vol. 5, no. 3 (Summer 1993): 21-31.

9. Information Processing -- ISO 7-bit and 8-bit Coded Character Sets -- Code Extension Techniques, ISO 2022:1986 (Geneva: International Organization for Standardization, 1986).

10. Information Processing, Open Document Architecture, ISO/IEC 8613:1989 (Geneva: International Organization for Standardization, 1989).

11. DEC Rdb, SQL Reference Manual (Maynard, MA: Digital Equipment Corporation, Order No. AA-PWQPA-TE, January 1993).

12. Multivendor Integration Architecture, Version 1.2 (Tokyo: Nippon Telegraph and Telephone Corporation, Order No. TR550001, September 1992).

13. R. Haentjens, "The Ordering of Universal Character Strings," Digital Technical Journal, vol. 5, no. 3 (Summer 1993): 43-52.

APPENDIX: SYNTAX OF Rdb VERSION 4.2

Format of CHARACTER SET Clause

```
<character data type> ::=
    <character string type>
        [ CHARACTER SET <character set specification> ]
  | <national character string type>

<character string type> ::=
```

```
    CHARACTER [ VARYING ] [ ( <length> ) ]
  | CHAR [ VARYING ] [ ( <length> ) ]
  | VARCHAR ( <length> )

<national character string type> ::=
    NATIONAL CHARACTER [ VARYING ] [ ( <length> ) ]
  | NATIONAL CHAR [ VARYING ] [ ( <length> ) ]
  | NCHAR [VARYING ] ( <length> )

<character set specification> ::=
        <character set name>

<character set name> ::= <name>
```

Character Set Names

```
                              DEC_MCS
                            | KANJI
                            | HANZI
                            | KOREAN
                            | HANYU
                            | DEC_KANJI
                            | DEC_HANZI
                            | DEC_KOREAN
                            | DEC_SICGCC
                            | DEC_HANYU
                            | KATAKANA
                            | ISOLATINARABIC
                            | ISOLATINHEBREW
                            | ISOLATINCYRILLIC
                            | ISOLATINGREEK
                            | DEVANAGARI
```

Example of CHARACTER SET

```
CREATE DATABASE FILENAME ENVIRONMENT
   DEFAULT CHARACTER SET DEC_KANJI
   NATIONAL CHARACTER SET KANJI
   IDENTIFIER CHARACTER SET DEC_KANJI;

CREATE DOMAIN NAMES_GENERAL CHAR(20);

CREATE DOMAIN NAMES_PRC CHAR(20)
   CHARACTER SET IS HANZI;

CREATE DOMAIN NAMES_MCS CHAR(20)
   CHARACTER SET IS MCS;

CREATE DOMAIN NAMES_KOREAN CHAR(20)
   CHARACTER SET IS HANGUL;

CREATE DOMAIN NAMES_JAPAN NCHAR(20);
```

Format of Literals

```
<character literal> ::=
    <character string literal>
  | <national character string literal>

<character string literal> ::=
    [ <introducer><character set specification> ]
    <quote>[ <character representation>... ]<quote>

<character representation> ::=
    <nonquote character>
  | <quote symbol>

<nonquote character> ::= !! See the Syntax Rules.

<quote symbol> ::= <quote> <quote>

<national character string literal> ::=
    N <quote>[ <character representation>... ]<quote>
```

Example of National Object Name

[EDITOR'S NOTE: The example of the national object name is in kanji and cannot be represented in the ASCII version.]

BIOGRAPHIES

Jim Melton A consulting engineer with Database Systems, Jim Melton represents Digital to the ANSI X3H2 Technical Committee for Database. He represents the United States to the ISO/IEC JTC1/SC21/WG3 Working Group for Database. He edited the SQL-92 standard and continues to edit the emerging SQL3 standard. Jim also represents Digital to the X/Open Data Management Working Group and to the SQL Access Group. Jim is the author of Understanding the New SQL: A Complete Guide, published in 1992, and is a regular columnist (SQL Update) for Database Programming & Design.

Hirotaka Yoshioka A senior software engineer in the International Software Engineering Group, Hiro Yoshioka is the project leader of the CDD/Repository/Japanese. He is a member of the internationalization special committee of ITSCJ (Information Technology Standards Commission of Japan) and ISO/IEC JTC1 SC22/WG20 internationalization. During the past nine years, he has designed and implemented the Japanese COBOL, the Japanese COBOL generator, and the internationalized DEC Rdb. Hiro joined Digital in 1984, after receiving an M.S. in engineering from Keio University, Yokohama.

TRADEMARKS

CDD/Plus, CDD Repository, DEC Rdb, Digital, and OpenVMS are trademarks of Digital Equipment Corporation.

PIC is a trademark of Wang Laboratories, Inc.

Unicode is a trademark of Unicode Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

X/Open is a trademark of X/Open Company Ltd.