Looking Forward to the Unicode Advantage: Internationalization and Integration

Michael Redford

Michael Redford studied German and Economics at Rutgers University (B.A.) and Theoretical Linguistics at the University of Constance (M.A.). He recently joined SAP AG as an Information Developer (Technology Development, Development Platforms — Internationalization). He can be reached at michael.redford@sap.com.

Unicode is an international character-encoding standard, and HTML, XML, and Java all harness Unicode to enable global, multilingual Internet data communication.[1] ABAP 6.10 joins the ranks of technologies that support Unicode, and with upcoming releases you can take advantage of Unicode-based mySAP solutions, which offer some decisive advantages, including the following:

- A Unicode system[2] can now take full advantage of XML and Java.
- Unicode-compliant ABAP paves the way for more efficient and effective integration of ABAP and Java applications.
- Unicode systems can be more tightly integrated with non-SAP products and offer a superior platform for collaborative, cross-system business applications.
- Last, but certainly not least, Unicode provides greater support for internationalization than was ever possible before. Unicode makes it possible to use all languages and language combinations without restrictions.

> *[Unicode] defines a consistent way of encoding multilingual text that enables the exchange of text data internationally and creates the foundation for global software. As the default encoding of HTML and XML, the Unicode Standard provides a sound underpinning for the World Wide Web and new methods of businesses in a networked world.*
>
> *– Introduction to The Unicode Standard, Version 3.0, p. 1*

[1] HTML 4.1 uses Unicode as the basic character set. However, earlier versions of HTML are based on ISO8859-1.

[2] An SAP installation that uses Unicode on the application server and in the database.

Unicode in a Nutshell

For those of you who may not be familiar with Unicode, it is an international standard that assigns characters a *unique* number within a single code page. Previously, there were different code pages for different language sets, so the same number represented a variety of characters depending on the code page being used. Unicode supports an enormous array of languages and scripts, including:

> Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Syraic, Thaana, Devanagari, Bengali, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Tibetian, Loa, Myanmar, Georgian, Hangul, Sinhala, Ethiopic, Cherokee, Khmer, Mongolian, Braille, CJK (Chinese, Japanese, Korean) ideographs, Hiragana, Katakana...

At first glance, some of these languages might seem unnecessary for your business environment. But take Bengali, for example. With more than 100 million speakers in India and Bangladesh, Bengali is one of the world's most widely spoken languages and therefore a potentially large market for some businesses. Even a language with a relatively small number of speakers can be critical to a business. If a company wants to set up a manufacturing center in a new country, for example, it will need a system

This article, the second in a two-part series,[3] will help managers, administrators, and developers assess whether the newer SAP Unicode systems fit their specific requirements, how these Unicode systems might offer key integration advantages, how a Unicode SAP system can be integrated into an existing landscape, and the key considerations when preparing to install a Unicode system from SAP.

Unicode is included as an enhancement to new mySAP component releases. Customers can use new releases without having to convert their SAP system to Unicode because the same 6.10 ABAP programs work in both Unicode and non-Unicode systems.[4] So, while using a Unicode system is entirely up to you, hopefully this article will convince you that Unicode makes mySAP.com an unbeatable integration platform, and will also prepare you for the day when you decide to flip the switch, so to speak, and leverage the Unicode functionality available to you on newer SAP systems (i.e., in database functions, applications, SAP GUI, etc.).

[3] The first article in this series, "Globalizing Applications Part 1: Pre-Unicode Solutions," appeared in the September/October 2001 edition of this publication.

[4] For more detail on Unicode and ABAP 6.10, refer to Gerd Kluger's article in the November/December 2001 issue of this publication.

Integrating a Unicode SAP System into Your Existing System Landscape

When you're ready to integrate a Unicode SAP system into your system landscape, you will find ample documentation, but the following section touches on a few central issues: RFC, SAP GUI, and external devices.

Starting with 6.10, RFC enhancements ensure that all data transfer between Unicode and non-Unicode SAP systems occurs smoothly.[5] All necessary data conversions occur in the Unicode system; therefore you do not need to make any changes to your existing non-Unicode systems. When configuring destinations on the Unicode system, you simply have to declare the RFC destination as a non-Unicode system, and then the data will be correctly converted with the appropriate code pages and language keys of the non-Unicode destination system.

Let us look at an example of how well a Unicode system integrates with non-Unicode systems, and

[5] The same holds true for data communication between SAP and non-SAP systems via BAPIs.

that can support the local language, regardless of how many speakers the language has. And a multinational company should be able to support the languages its *customers* use, in order to tailor its products and marketing for each language in its market and to be able to print orders and invoices in the customer's language.

The sheer size of the Unicode code page makes it impossible to show all of the characters, but the figure below shows you how the characters are organized, and the major areas are indicated as follows:

- The uppermost portion contains all 7-bit ASCII characters, followed by all of the characters used in scripts with an alphabet, such as English, Georgian, Arabic, Hebrew, Hindi, Malayalam, etc.
- The next band contains various symbols: mathematical operators, technical symbols, the ubiquitous dingbats, etc.
- The largest section (CJK ideographs) contains ideographs used in Chinese, Japanese, and Korean.
- All of the Hangul characters and character combinations used in Korean are in the Hangul section.
- The "surrogate area" was designed to ensure that there is adequate room for future additions to the Unicode Standard, which is continuously growing (additional languages and symbols, characters from dead languages, etc.).
- The "compatibility area" is a repository for various groups of characters, including characters that are used in non-Unicode code pages, but need to be defined separately in order for Unicode to be compatible with non-Unicode code pages. For example, some special half-width and double-width characters from the Japanese code page Shift-JIS (SJIS) are in this area.

(As a convention, the number is written in hexadecimal notation and prefixed with a "U+". /A/, for example, is U+0041 and /가/ is U+AC00.)

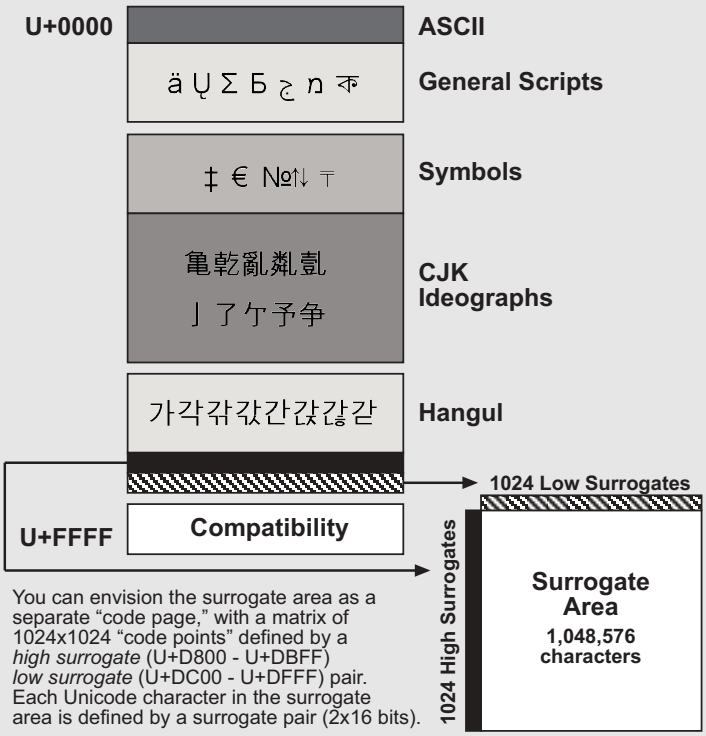
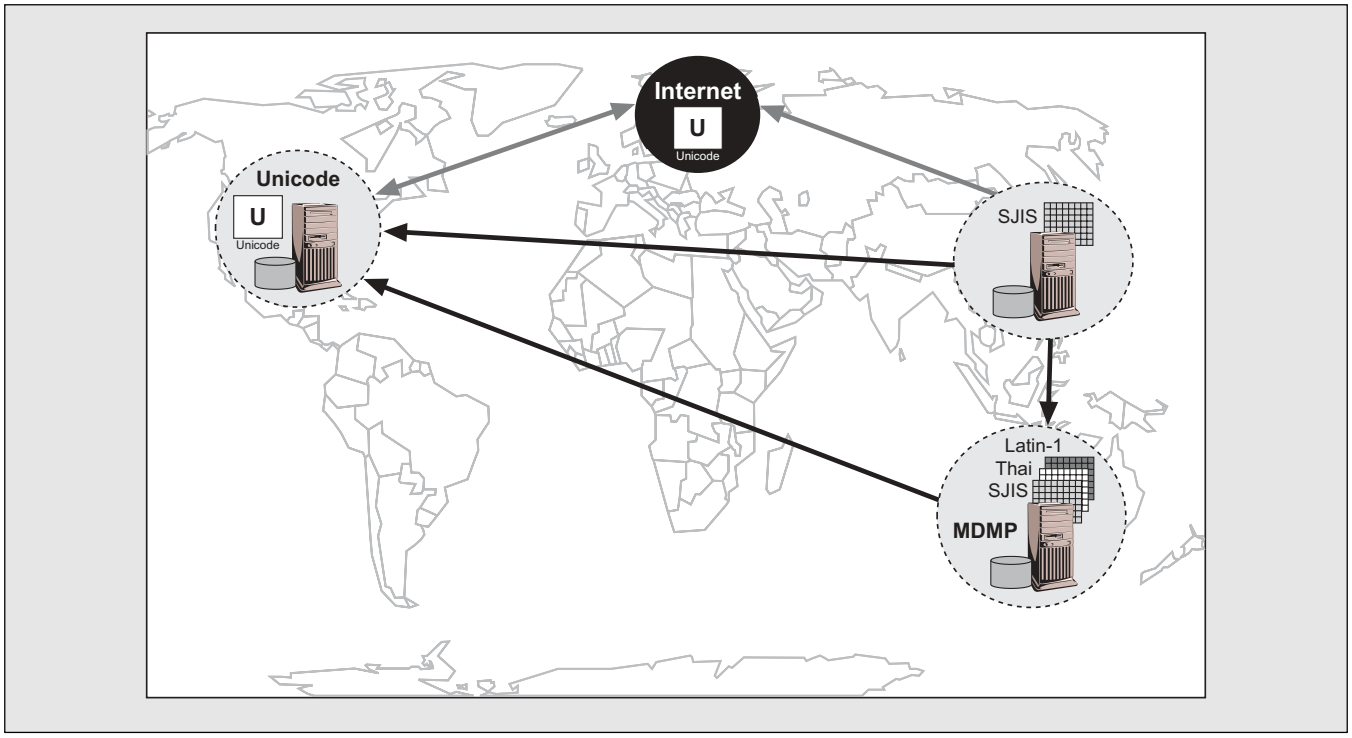


Figure 1 System Communication Between Unicode and Non-Unicode SAP Systems



the advantages of using a Unicode system. A multinational company — Widgets Unlimited — has a Unicode SAP system at its headquarters in the US, a single code page SAP system in Japan (Shift-JIS, or SJIS, which contains Japanese and English characters), and an MDMP SAP system in Australia (Latin-1/SJIS/Thai).[6]

Figure 1 shows the communication between the three locations. Arrows indicate unproblematic data paths (note that 7-bit ASCII characters can always be sent and received and are therefore not indicated). In this example, Japanese and English (ASCII) data can be sent and received by all offices, but the Japanese office cannot receive any data with Thai characters from the Australian office, because SJIS does not contain those characters. The same restricted communication occurs regardless of how the data is sent: All data sent through the Internet is encoded in Unicode, but that does not mean that a non-Unicode system can correctly deal with the text data it receives. To truly be able to communicate without any code page incompatibilities, Unicode is the only solution.

In summary, a Unicode system can be integrated within an existing landscape of non-Unicode systems, and moreover, it can always communicate with all other systems, regardless of the code page in the non-Unicode system.

SAP GUI Unicode Support

All SAP GUIs (HTML, Java, Windows) support Unicode alongside all the non-Unicode code pages already supported.[7] Because SAP GUI is backward-compatible, a single SAP GUI can be used to access both Unicode and non-Unicode systems, and therefore only one GUI is needed per frontend. Revisiting Widgets Unlimited will show exactly how to use Unicode. As shown in **Figure 2**, a user at the US

[6] For a detailed explanation of code pages and Multi-Display/Multi-Processing (MDMP), see my previous article in the September/October 2001 issue of this publication.

[7] Support for Unicode is feasible with the 6.20 SAP GUI, and will be improved in later releases.

Code Pages Are the Problem!

A code page defines how a byte sequence is mapped to a character; put differently, a code page determines what a byte sequence "stands for." As an analogy, what does "BA" stand for? It stands for "Bachelor of Arts" or "British Airways," depending on the context. Similarly, the byte sequence "BA" can also stand for several characters, depending on the code page (see the table below).

Byte	Code Page	Character
0xBA	ISO8859-1	º
0xBA	ISO8859-2	ş
0xBA	ISO8859-4	ē
0xBA	ISO8859-5	Ќ
0xBA	ISO8859-7	΄Ι
0xBA	ISO8859-8	÷
0xBA	SJIS	ｺ

As soon as more than one code page is used, code points become ambiguous: a code point can define one character in one code page and a completely different character in another code page. But no single code page is sufficient, so you need more than one code page — that is, until Unicode.

For more information on code pages, see my previous article "Globalizing Applications Part 1: Pre-Unicode Solutions," in the September/ October 2001 issue of *SAP Professional Journal*.

Figure 2 Select Unicode, and All Unicode Characters Become Available

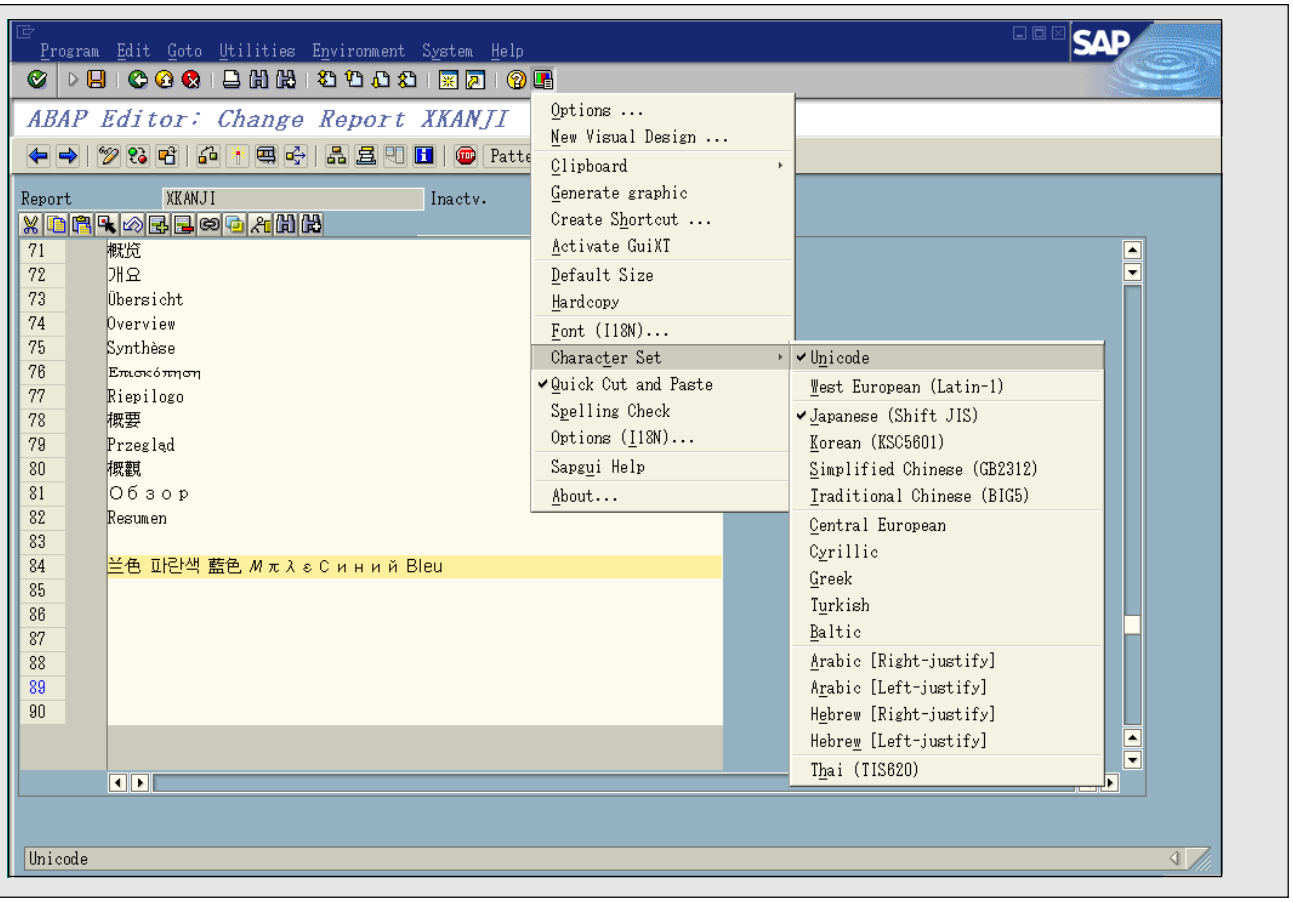
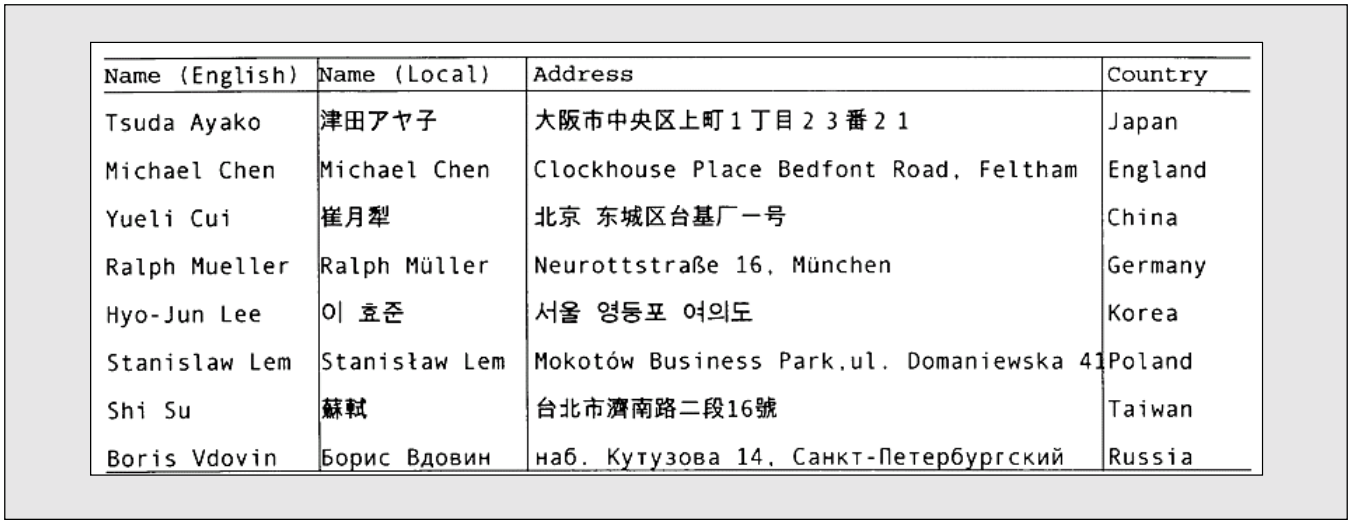


Figure 3 Unicode Printout (Lexmark)



Name (English)	Name (Local)	Address	Country
Tsuda Ayako	津田アヤ子	大阪市中央区上町１丁目２３番２１	Japan
Michael Chen	Michael Chen	Clockhouse Place Bedfont Road, Feltham	England
Yueli Cui	崔月犁	北京 东城区台基厂一号	China
Ralph Mueller	Ralph Müller	Neurottstraße 16, München	Germany
Hyo-Jun Lee	이 효준	서울 영등포 여의도	Korea
Stanislaw Lem	Stanisław Lem	Mokotów Business Park,ul. Domaniewska 41	Poland
Shi Su	蘇軾	台北市濟南路二段16號	Taiwan
Boris Vdovin	Борис Вдовин	наб. Кутузова 14, Санкт-Петербургский	Russia

office simply selects Unicode as the character set and the full span of Unicode characters becomes available. All characters can be seen *simultaneously* in a user interface, and it is possible to enter multiple languages in a single field.

External Devices and Unicode

Printers and other external devices that SAP currently supports will continue to be supported on Unicode and non-Unicode systems, and therefore no additional hardware is needed. But consider purchasing a Lexmark printer that supports Unicode, because a single Unicode printer can produce high-speed, high-quality printouts in any language.[8] With Unicode printers, the days of language-specific printer code pages and hardware become a thing of the past, which can reduce the number of printers a company needs. Fewer printer purchases, installations, and maintenance contracts can translate into considerable cost savings. Note that you can use a Unicode printer with your current system (MDMP as well!).

In our ongoing example, Widgets Unlimited can print out a list of all its customers in their native languages, as you can see in **Figure 3**.

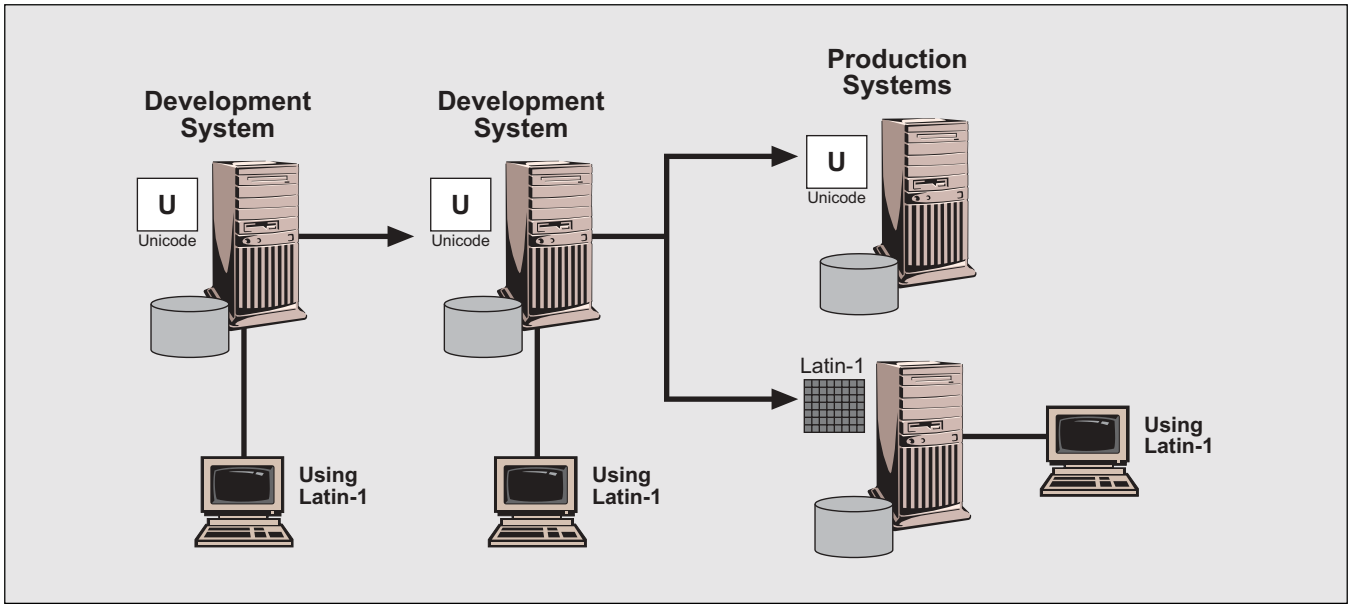
[8] Right now, Lexmark is the only manufacturer that produces such printers.

Unicode in the Development and Test System Landscape

The same API is used for both Unicode and non-Unicode systems, so Unicode libraries can communicate with non-Unicode systems. In addition, both `R3trans` and `R3load` can be used between your Unicode and non-Unicode systems, which allows R/3 objects, such as customizing data, to be transported into and out of Unicode and non-Unicode systems. You therefore do not need separate development and test systems for Unicode and non-Unicode systems. I recommend that you use a Unicode development and test system even if the production system will use another system code page, because of improvements to ABAP 6.10 syntax, as I will discuss in a moment.

Of course, there is a danger that Unicode characters in the test system could be undefined in the code page of a non-Unicode productive system. But if you set the frontend code pages of these systems to match the system code page of the production system, only those characters that are defined in the productive system will be entered into the development or test system. **Figure 4** shows such a system landscape. The frontend code page for the future production system (Latin-1) is used in the development and test systems; this prevents non-Latin-1 characters from inadvertently being entered into the system.

Figure 4 Unicode Development and Test Systems: Landscape for All Production Systems



The Unicode Advantage: Integration Scenario

In the September/October 2001 issue of *SAP Professional Journal*, I provided a brief introduction to code pages and sketched the two internationalization solutions SAP currently offers: blended code pages and Multi-Display/Multi-Processing (MDMP). Customers have successfully implemented these systems, and they provide SAP users with the ability to work in their own native languages within an SAP system. But at the same time, blended code pages and MDMP are SAP technologies, and non-SAP software cannot use or comprehend either of them.

In a Unicode SAP system, all data is in a standard, open format that non-SAP systems *can* access, which promotes the integration of mySAP components in mixed environments. In addition, ABAP 6.10 and Java use the same character encoding (UTF-16, which is described in the section "Step 1: Determine Whether You Will Need Additional Hardware"). This enables local communication via shared memory, with greatly improved performance.

Now let's expand the scope of our earlier example and consider a collaborative scenario. Widgets Unlimited's Japanese office has just started working with a new manufacturer in Germany and uses XML to communicate with the manufacturer's non-SAP system. A customer can place his order directly on the manufacturer's portal and can view the shipment status, although the distributor is responsible for shipment and invoicing. The customer can check the status of the order at any time. Let us assume that all communication between the parties is via XML.

In a non-Unicode system, the Japanese customer would have to somehow determine which languages to use when entering the order. If both the German manufacturer and the Japanese distributor have a

(continued on next page)

Preparing for a Unicode Installation

A Unicode SAP system requires a Unicode database, a Unicode component, and an SAP GUI that supports Unicode (see the earlier section "SAP GUI Unicode Support"). **Figure 5** summarizes the platform/database combinations that SAP supports.

To provide Unicode locales (locales specify a user's country and language to determine sorting and the formatting of items such as numbers, time and date, and currencies), SAP uses and provides the International Components for Unicode (ICU). The ICU is a C/C++ and Java library that provides many internationalization functions, including locales, transliteration, and language-sensitive sorting (for more information on the ICU, visit **http://oss.software.ibm.com/icu/**).

Tentatively, Unicode-based mySAP.com will be available in the following component releases:

- CRM 3.1
- BW 3.0B
- R/3 Enterprise
- APO release following 3.1

(continued from previous page)

non-Unicode system with the Shift-JIS (SJIS) code page for Japanese, then the customer could enter Japanese text. However, while it is likely the Japanese distributor supports Japanese text, it is unlikely that the German manufacturer does. In this case, the customer would get an error message, such as "Please place your order in English, our backend system does not support Japanese." Since the customer and distributor are both in Japan, it makes no sense to force them to use English, but the limitations of non-Unicode code pages have forced all parties to use 7-bit ASCII (i.e., English) for communications. Note also that since the distributor is in Japan, the German manufacturer does not need to read the shipping information during order processing, which counters any arguments that English should always be used in international settings.

You may be saying to yourself, "What is the problem? They are using XML!" Actually, it doesn't matter — in this case, XML does not solve the problem. Without a Unicode system, XML data transfer can fail *completely* if the sender and receiver do not use the same code page. Thus language-specific characters, technical symbols, and currency symbols will not be transferred, leading to data loss and even to process failures.*

With a Unicode system, the Japanese customer enters his order in Japanese. When he later logs on to the system to check the status of his order and the estimated delivery date, the information is again given in Japanese. Unicode ensures data flow across all of these boundaries, but at the same time enables that data to be more than just 7-bit ASCII. Collaboration only works if there is no data lost during transmission, and the more participants involved in collaboration, the greater the chance that code page incompatibilities lead to data loss. ASCII data can always be sent and received, but in a global setting, such restrictions only hamper collaboration.

* In addition to language-specific characters, the Unicode Standard also includes technical and currency symbols, including the Euro symbol €.

Figure 5 Supported Platform/Database Combinations

Database	Platform							
	Win2000*	Linux*	Solaris	HP-UX	Tru64	AIX	OS/400	OS/390
SQL Server	✓	—	—	—	—	—	—	—
Oracle	✓	✓	✓	✓	✓	✓	—	—
DB/2	✓	✓	✓	✓	—	✓	✓	—**
SAP DB	✓	✓	✓	✓	✓	✓	—	—

* Supports 32-bit systems, and will support 64-bit versions in 2002.
** OS/390 support is planned for 2003 with DB/2 8.0.

Let's now walk through the three steps involved in preparing for a Unicode installation:

1. Determine whether you will need additional hardware.

2. Check that your ABAP programs comply with ABAP 6.10 syntax by using transaction UCCHECK to find the lines that must be modified. Modify any programs that are not compliant. For all programs, set the attribute "Unicode enabled." Adapt any C/C++ programs using the ccQ program.[9]

3. Perform runtime tests for Unicode compliance and use the Coverage Analyzer (transaction SCOV) to monitor the testing.

We'll now take a closer look at each of these tasks in turn.

Step 1: Determine Whether You Will Need Additional Hardware

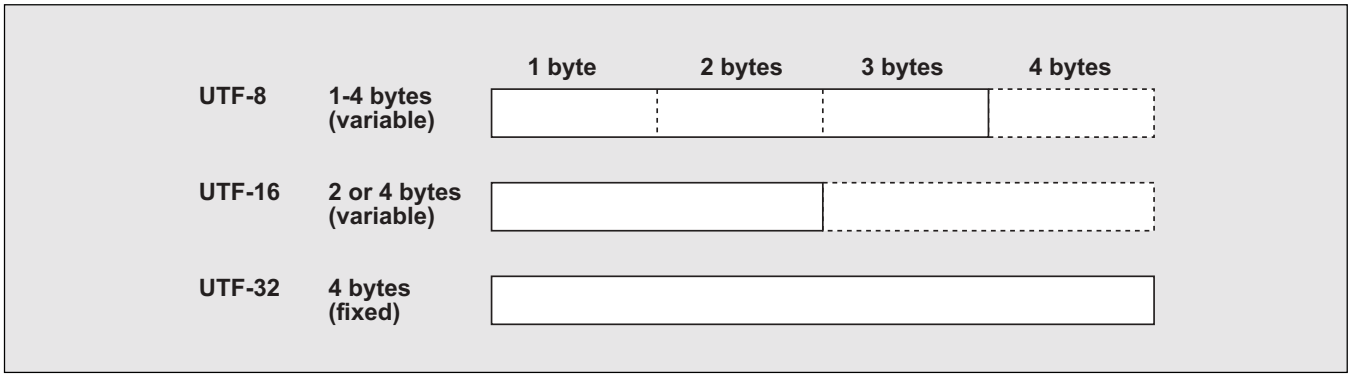
The size of a character in a Unicode system is larger than it is in a non-Unicode system, and the added length can lead to slightly greater hardware demands. SAP estimates that customers with an existing Latin-1 system will need an additional 30% CPU capacity for their Unicode system, and customers with an MDMP system will require no additional CPU capacity because a Unicode system does not require multiple code page handling. These estimates are based on initial tests; additional tests are under way on different platforms to provide further data. The additional memory requirement is estimated to be 30%, based, again, on initial tests. Both performance and hardware consumption optimization are in progress.

It should also be mentioned that Unicode offers an additional advantage over an MDMP system with many languages. In an MDMP system, English user data has to be copied into all languages to make it visible for non-English users. This creates redundancies in the database and leads to additional maintenance. With Unicode everyone can log in as an English user and read and write in any script they want. In this context, it should also be mentioned that these requirements are minimal when compared to the demands that Java applications place on system hardware.

The amount of additional database capacity depends on several factors, and to understand what they are, we have to take a closer look at how Unicode works. Each Unicode character has a unique number. For example, /A/ is U+0041 and /가/ is U+AC00 (recall that "U+" is a naming convention, and the number is in hexadecimal notation). There

[9] This program alters library functions that handle character types from the C standard library. More documentation is available via the ftp sites **ftp.sap.com/pub/i18N/ccQ** and **ftp.sap.com/pub/i18N/utf16**.

Figure 6 Unicode Encoding Schemes



are three ways to encode these characters, as defined by the Unicode Transformation Format (UTF):[10]

- **UTF-8 (8-bit):** With this encoding, characters do not have a fixed length. ASCII characters, such as /A/, are one byte long, while other characters, such as /Ä/, are two bytes long and characters from Asian languages require three bytes. Surrogates are four bytes long. (At the moment there are only a few characters in the surrogate area.)

- **UTF-16 (16-bit):** With this encoding, characters (except those in the surrogate area) are two bytes long. In the surrogate area they are four bytes long.

- **UTF-32 (32-bit):** With this encoding, all characters are four bytes long.

The UTF encoding schemes are illustrated in **Figure 6**. (Dotted lines indicate the length of a "surrogate character.")

In addition to the three UTF encoding schemes, there is an additional "Compatibility Encoding Scheme of UTF-16 on an 8-bit base" (CESU-8). This encoding scheme is equivalent to UTF-8, except that the characters in the surrogate area are handled differently (the motivations for CESU-8 require a technical discussion that is beyond the scope of this article).[11] The major database manufacturers offer Unicode databases using either CESU-8 or UTF-16.

Figure 7 contains a list of database manufacturers, the encoding scheme used on their databases, and the estimated additional storage space requirements for the databases.

The additional storage space required will also depend on the nature of the data in your databases. For example, the character /A/ and all other 7-bit ASCII characters are one byte long in non-Unicode code pages and in CESU-8, but in UTF-16 all ASCII characters are two bytes long. UTF-16 therefore *doubles* the storage space required for all ASCII data. For Asian languages, the reverse is true: UTF-16 requires less storage space than CESU-8.

Figure 8 compares character lengths in different languages and encodings.

[10] Although it may sound like there is more than one Unicode system, this is not the case! Transforming one encoding into the other is a simple mathematical process, and since exactly the same characters are in all encodings, no character corruption can occur converting between UTFs. The difference lies merely in how many bytes define a Unicode character.

[11] CESU-8 will be registered with the Internet Assigned Numbers Authority (IANA), and is already in use at several companies (for more information, see the published *Proposed Draft Unicode Technical Report #26*, which can be found on the Unicode homepage at **www.unicode.org/unicode/reports/tr26/**).

Figure 7 Unicode Encodings and Additional Storage Space Required

Manufacturers	Encoding	Additional Storage Required
Oracle, DB/2 (AIX), SAP DB (8.0)	CESU-8/ UTF-8	10-20%
SQL Server, DB/2 (AS/400), SAP DB (7.0)	UTF-16	60-70%

Figure 8 Character Length (Without Surrogates)

ISO8859 Languages (CESU-8<UTF-16)

Encoding	/A/	/Ä/
ISO8859-1	1 byte	1 byte
CESU-8	1 byte	2 bytes
UTF-16	2 bytes	2 bytes

Asian Languages (CESU-8>UTF-16)

Encoding	/A/	/晒/
SJIS	1 byte	2 bytes
CESU-8	1 byte	3 bytes
UTF-16	2 bytes	2 bytes

Note that CESU-8 does require some additional space for non-Asian characters as well, because characters such as /Ä/ are two bytes long in CESU-8, as compared with one byte in ISO8859 code pages.

Because the amount of additional hardware is dependent on several variables, the additional space required will vary, but tools are being developed to aid customers in approximating the additional storage space needed, if any.

Step 2: Check That Your ABAP Programs Comply with ABAP 6.10 Syntax

Prior to 6.10, the length of a character in ABAP was, by definition, one byte. This equivalency and the treatment of structures as character fields allowed implicit typecasts. For example, it was possible to write the following:

```
FORM f USING p1 p2.
  DATA: len TYPE I.
  DESCRIBE FIELD p1 LENGTH len.
  DESCRIBE DISTANCE BETWEEN p1
AND p2 INTO len.
ENDFORM.
```

With Unicode, one character is no longer one byte, and references to characters cannot be based on byte length. Whenever implicit or explicit reference is made to the internal length of a character in ABAP, the specification `IN CHARACTER MODE` or `IN BYTE MODE` must be added to define the unit of measure. ABAP 6.10 has a more abstract definition of a character and such fragile, memory-layout-oriented programming with structures is no longer possible.

Version 6.10 of the ABAP language complies with the Unicode Standard, and character-like

Figure 9 Unicode-Enabling Example: Unmodified Pre-Release 6.10 Code

```
CLASS CL_ABAP_CHAR_UTILITIES DEFINITION LOAD.

CONSTANTS: HEX0(1) TYPE X VALUE '00',
           CRLF(2) TYPE X VALUE '0D0A'.

DATA: BEGIN OF LINE1,
        TEXT1(10)  TYPE C VALUE 'system:   ',
        MARK1(1)   TYPE X VALUE HEX0'
        TEXT2(10)  TYPE C VALUE 'user:     ',
        MARK2(1)   TYPE X VALUE HEX0,
        TEXT3(10)  TYPE C VALUE 'client:   ',
        MARK3(1)   TYPE X VALUE HEX0,
        SPACE(100) TYPE C,
      END OF LINE1,
      LINE2(133) TYPE C.

REPLACE: HEX0 WITH SY-SYSID INTO LINE1,
         HEX0 WITH SY-UNAME INTO LINE1,
         HEX0 WITH SY-MANDT INTO LINE1.

CONDENSE LINE1.

CONCATENATE LINE1 CRLF INTO LINE1.

LINE2 = LINE1.

WRITE: / LINE2.
```

Unicode Error!

HEX0 is hard-coded '0D0A', and is platform-dependent. Should be type C.

data (i.e., ABAP types C, N, D, T) are defined as UTF-16 Unicode characters. To reduce memory requirements, UTF-16 has been implemented instead of UTF-32 for internal representation, which offers a compromise between complexity and memory usage. Moreover, UTF-16 is the standard used in Java and Visual Basic, and by adopting the same encoding internally, the same buffers can be shared. Note that this is the representation of a character used in ABAP, which is not related to the encoding of the database (see also footnote 10).

Below is a partial list of further changes in ABAP that result from the stricter Unicode syntax:

- **Move:** Flat structures are no longer treated as C fields for conversions.
- **String:** String processing statements are strictly divided into character-processing statements and byte-processing statements.
- **Offset:** Offset- or length-based access to structures is only permitted if the structures are flat and the offset/length specification includes only character type fields from the beginning of the structure.
- **Dataset:** If reading or writing to text files with READ DATASET and TRANSFER, the character encoding must be explicitly specified.
- **Others:** Obsolete statements are no longer allowed if the key is not entirely of character type, e.g., LOOP AT dbtab, READ TABLE dbtab, and READ TABLE itab.

Figure 10 Unicode-Enabling Example: Modified Code for Release 6.10 and Later

```
CLASS CL_ABAP_CHAR_UTILITIES DEFINITION LOAD.

CONSTANTS:
  HEX0(1) TYPE C VALUE CL_ABAP_CHAR_UTILITIES=>MINCHAR,
  CRLF(2) TYPE C VALUE CL_ABAP_CHAR_UTILITIES=>CR_LF.

DATA:
  BEGIN OF L_LINE1,
    TEXT1(10)  TYPE C VALUE 'system:   ',
    MARK1(1)   TYPE C VALUE HEX0,
    TEXT2(10)  TYPE C VALUE 'user:     ',
    MARK2(1)   TYPE C VALUE HEX0,
    TEXT3(10)  TYPE C VALUE 'client:   ',
    MARK3(1)   TYPE C VALUE HEX0,
    SPACE(100) TYPE C,
  END OF L_LINE1,
  L_LINE2(133) TYPE C.

REPLACE: HEX0 WITH SY-SYSID INTO L_LINE1,
         HEX0 WITH SY-UNAME INTO L_LINE1,
         HEX0 WITH SY-MANDT INTO L_LINE1.

CONDENSE L_LINE1.

CONCATENATE L_LINE1 CRLF INTO L_LINE1.

L_LINE2 = L_LINE1.

WRITE:/ L_LINE2.
```

Class defines low-level constants.

HEX0 is type C.

To provide Unicode support, SAP chose not to use a separate Unicode data type (in standard ABAP program code), and thus Unicode-enabling is transparent. As a result, there is no conversion between Unicode and non-Unicode characters in ordinary application coding. This transparency has a great advantage: the same 6.10 ABAP source code runs on both Unicode and non-Unicode systems. Customers can therefore use Unicode and non-Unicode systems in parallel — simply recompile and run.

In a Unicode system, all ABAP programs must be 6.10-compliant, including programs written by customers and partners. ABAP 6.10 programs will run on both Unicode and non-Unicode systems, but ABAP programs that are not 6.10-compliant (i.e., Unicode-compliant), will not run properly in a Unicode system. Existing coding may therefore have to be partially adapted to these restrictions in order to be executable in a Unicode environment.

I strongly recommend adopting these new restrictions even in a non-Unicode system, because they will improve the quality of all existing programs. The new restrictions that Unicode places on ABAP programming have a further advantage (aside from Unicode): stricter type definitions make ABAP programs more efficient, and they eliminate a source for bugs. In addition, checking existing code for 6.10 compliance will likely reveal errors in existing programs that had not been noticed, and that are not directly related to Unicode.

Figure 9 and **Figure 10** show a “before-and-after”

example of Unicode-enabling an ABAP program. The programming style used in the examples is not optimal, either before or after, but it shows that small adjustments can make existing programs ready for a Unicode world. In Figure 9, you see that the coding contains type X fields — these fields cannot be used to store characters, so the code must be modified, as shown in Figure 10.

For more on Unicode and ABAP 6.10, see also the section "The New Release 6.10 Approach" in Gerd Kluger's article "File I/O with ABAP — Problems, Workarounds, and Prudent Practices" (*SAP Professional Journal*, November/December 2001).

Step 3: Perform Runtime Tests for Unicode Compliance and Monitor the Testing

You will need to test your programs to make sure that they are ABAP 6.10-compliant. SAP developed two tools to make sure that all SAP programs are Unicode-compliant, and the same tools are at your disposal: transaction UCCHECK and the Coverage Analyzer, both of which are available as of Release 6.10.[12] SAP used these tools to Unicode-enable its own programs, and based on SAP's firsthand experience, roughly 10,000 lines of ABAP code can be made compliant per developer per day. Developers will need to take the following steps:

1. **Run transaction UCCHECK.** This transaction scans existing programs to find any code that is non-Unicode-compliant. It checks the syntax of all ABAP programs and indicates lines of code that need to be modified, as well as lines that cannot be analyzed statically. The results list returned by this transaction contains code that is not compliant, along with an error message. In **Figure 11**, you see the result of running UCCHECK on several ABAP programs.

2. **Modify any programs** that are identified as non-compliant. Refer back to Figures 9 and 10 for a "before-and-after" demonstration of how code must be modified.

3. **Set the "Unicode enabled" attribute** for each ABAP program after you have modified them. There are two ways to do this:

   - In transaction UCCHECK, you can select all of the programs that you want to enable and then select "Set Unicode Attribute" from the toolbar.
   - Go to transaction SE38, enter the name of the program, select the radio button "Attributes," and then select "Change." Then select the checkbox "Unicode checks active."

> ***Warning!***
>
> *Only ABAP programs with the Unicode attribute set are executable in a Unicode system. All changes should be tested.*

After developers have set "Unicode enabled," all programs should be tested. Quality managers can use the Coverage Analyzer to make sure that the testing is carried out.

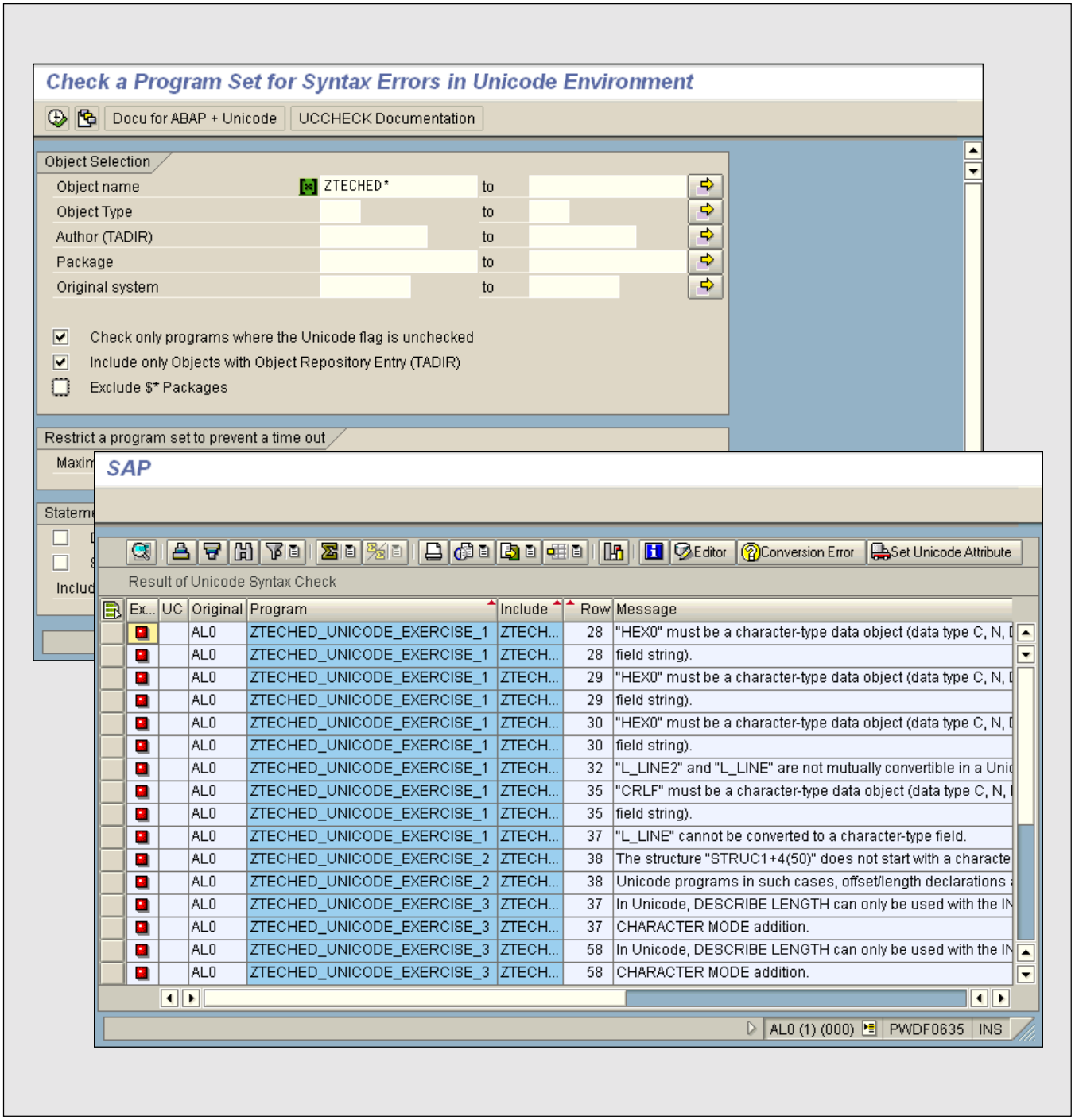
4. **Run the Coverage Analyzer** (transaction SCOV), a system diagnostic tool you can use to ensure that all 6.10-compliant programs have been tested.

> ***✓ C/C++ Tip***
>
> *Should you have any C/C++ programs, they will require some modifications as well, since characters must be defined as Unicode, i.e., type charU. Fortunately this task can be done almost entirely automatically, using the program ccQ (see footnote 9).*

With the Coverage Analyzer, quality managers are able to monitor the progress of a Unicode-

[12] For some tips on how to start preparing for Unicode with a 4.6 or earlier release, see "Tips for Developers" on page 111.

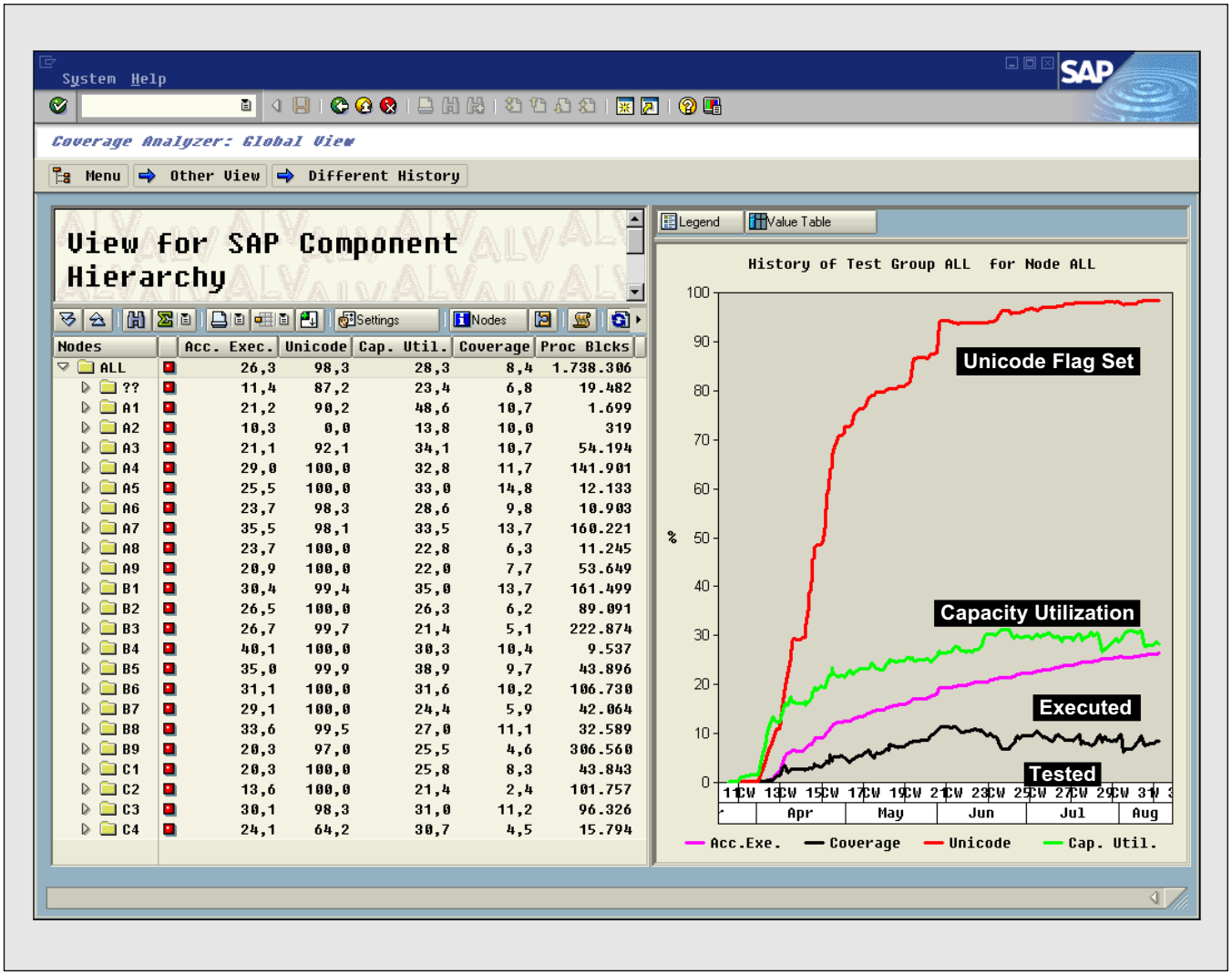
Figure 11 Transaction UCCHECK (with Results List)



enabling project. But how does transaction SCOV actually monitor testing?

ABAP programs consist of program units such as form routines, function modules, and events such as `AT USER-COMMAND`. These units are called *processing blocks*. A program is considered tested if all processing blocks are

Figure 12 *Coverage Analyzer: Global View*



Nodes	Acc. Exec.	Unicode	Cap. Util.	Coverage	Proc Blcks
ALL	26,3	98,3	28,3	8,4	1.738.306
??	11,4	87,2	23,4	6,8	19.482
A1	21,2	90,2	48,6	10,7	1.699
A2	10,3	0,0	13,8	10,0	319
A3	21,1	92,1	34,1	10,7	54.194
A4	29,0	100,0	32,8	11,7	141.901
A5	25,5	100,0	33,0	14,8	12.133
A6	23,7	98,3	28,6	9,8	10.903
A7	35,5	98,1	33,5	13,7	160.221
A8	23,7	100,0	22,8	6,3	11.245
A9	20,9	100,0	22,0	7,7	53.649
B1	30,4	99,4	35,0	13,7	161.499
B2	26,5	100,0	26,3	6,2	89.091
B3	26,7	99,7	21,4	5,1	222.874
B4	40,1	100,0	30,3	10,4	9.537
B5	35,0	99,9	38,9	9,7	43.896
B6	31,1	100,0	31,6	10,2	106.730
B7	29,1	100,0	24,4	5,9	42.064
B8	33,6	99,5	27,0	11,1	32.589
B9	20,3	97,0	25,5	4,6	306.560
C1	20,3	100,0	25,8	8,3	43.843
C2	13,6	100,0	21,4	2,4	101.757
C3	30,1	98,3	31,0	11,2	96.326
C4	24,1	64,2	30,7	4,5	15.794

marked as executed and no runtime errors occurred.[13]

Figure 12 shows the global view of the Coverage Analyzer, which quality managers can use to determine the extent of a Unicode-enabling project over time. The graph in Figure 12 shows the relevant statistics over a four-month period. **Figure 13** summarizes these statistics. Based on the low percentages of executed and tested code blocks, the quality manager for this test system will probably want to push for more testing.

After the desired level of testing has been reached, all customer programs will be ready for Unicode, and they will run in non-Unicode systems as well, of course.

In this last section, I walked you through the nuts and bolts of how to make sure that your programs are Unicode-compliant, but the process itself is straightforward:

[13] SCOV does not provide details of a program's flow logic (e.g., branches of an `if-then-else` construction), and it does not consider possible changes of parameter types during different form routine calls, but these features may be added in later releases.

Figure 13 Summary of Statistics for a Four-Month Period

Type of Testing	Percent Tested	Description
Acc. Exe. (Executed)	26%	Processing blocks executed since the start of the Coverage Analyzer
Unicode (Unicode Flag Set)	98%	Processing blocks with the Unicode flag set
Cap. Util. (Capacity Utilization)	29%	Ratio of used processing blocks to loaded processing blocks
Coverage (Tested)	8%	Processing blocks executed in active version without runtime error

Tips for Developers

To expedite Unicode-enabling before 6.10:

- ✓ Avoid using information about memory layout.
- ✓ Avoid implicit typecasts by using correct structure types when moving or comparing structures.
- ✓ Assign parameters a concrete type rather than leaving them type-less.

1. Set up a 6.10 development system.
2. Run UCCHECK and modify any programs that are not Unicode-compliant.
3. Set the "Unicode enabled" attribute for your ABAP programs, and run and monitor tests of those programs with the Coverage Analyzer.

Conclusion

Unicode uniquely defines every character, and it is the character encoding used by modern standards, including HTML and XML, and in programming languages such as Java, JavaScript, and ABAP 6.10. In ABAP, a more abstract character representation was adopted to support Unicode, and this also makes ABAP code more efficient. At the same time, this strategy enables the same code to work in both Unicode and non-Unicode systems. To ensure that a Unicode system integrates into existing R/3 landscapes, a single frontend GUI can access both Unicode and non-Unicode systems, and the same development and test systems can be used for both Unicode and non-Unicode productive systems.

In a Unicode system, all programs must be 6.10-compliant, which requires that some existing customer and partner programs be altered: the syntax changes result in cleaner, more efficient code. To help in this process, SAP developed programs to identify the code that has to be altered, and to monitor testing of that code.

This article showed you the advantages of Unicode, which can be summarized as follows:

✓ **Better integration with non-ABAP programs and seamless integration with existing SAP systems.** A Unicode system can communicate with legacy R/3 and mySAP components, which protects your investment while offering additional functionalities necessary for your organization to collaborate and compete. With Unicode-compliant

ABAP 6.10, SAP components can truly take advantage of Internet technologies, and this also paves the way for greater integration of R/3 with Java.

✓ **Assured data transfer.** Collaboration only works if there is no data lost during transmission. Without Unicode, XML data transfer can fail *completely* if the sender and receiver do not use the same code page. The more participants involved in collaboration, the greater the chance that code page incompatibilities can lead to data loss. ASCII data can always be sent and received, but in a global setting, such restrictions only hamper collaboration. As the standard encoding for all Internet communication, Unicode ensures cross-application data exchange, regardless of the system landscape.

✓ **Enhanced internationalization.** Unicode specifies a unique number for virtually all characters used by languages spoken in the world, and with Unicode there are no restrictions on which languages and language combinations can be used. Unicode allows companies to install a central system for worldwide business processes, e.g., to gather aggregate corporate data.

Acknowledgements

I would like to thank all of my colleagues at Development Platforms — Internationalization for their comments and suggestions, which helped improve and clarify many points in this article. At the same time, all errors are my own.

