Muhammad F. Mudawwar
American University in Cairo

Multicode: A Truly Multilingual Approach to Text Encoding

Unicode was designed to extend ASCII for encoding text in different languages, but it still has several important drawbacks. Multicode overcomes those drawbacks.

The Internet's explosive growth and the recent interest in building international software appropriate for use in different countries have increased demand for a standard character set for use with many different languages. Currently, the ASCII character set[1] is the world's most widely accepted and used standard character set for computers, operating systems, compilers, and e-mail systems. However, while ASCII encoding adequately represents English text, it does not address the problem of handling text in other languages.

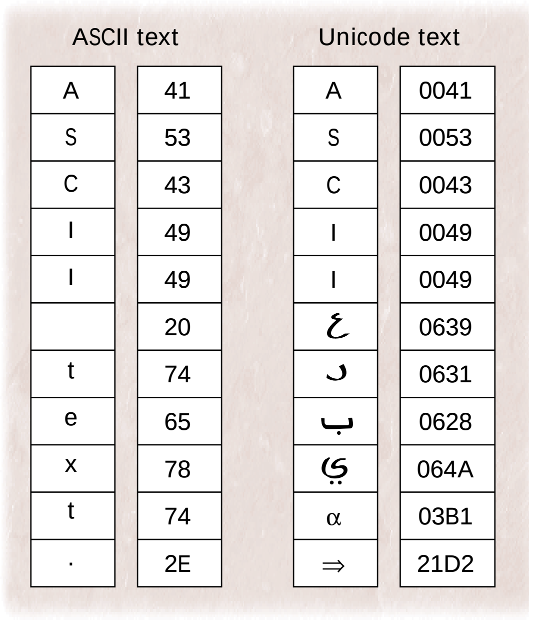
ASCII is a 7-bit code and defines only 128 characters. When used with an 8-bit character format, the 128 characters that ASCII would not use could be used as extensions to ASCII. These extensions would be used to define characters of different languages. For example, the ISO 8859 standard[2] defines a Latin extension (that supports many European languages), as well as Cyrillic, Arabic, Greek, Hebrew, and other language extensions.

To handle documents that mix English with a second language, you must use the ASCII extension for the second language. However, this approach presents two key problems:

- The 128 characters represented by the extensions are used for every language besides English. Therefore, mixing more than one language with English presents a problem: You cannot tell which language they refer to in a piece of text. Traditionally, programs used awkward escape sequences to shift the meaning of bits.
- The 8-bit extension cannot accommodate languages with more than 128 characters, such as Japanese, which would require a 16-bit extension.

Major computer companies and organizations developed the Unicode character set standard to address these problems. However, the 16-bit Unicode system, which uses one character set for all languages, creates some new problems. I developed Multicode, which uses multiple character sets for multiple languages, to address Unicode's principal drawbacks.

The Multicode system is new, so it needs more study and more exposure to the computer industry before



ASCII text		Unicode text	
A	41	A	0041
S	53	S	0053
C	43	C	0043
I	49	I	0049
I	49	I	0049
	20	ع	0639
t	74	ر	0631
e	65	ب	0628
x	78	ي	064A
t	74	α	03B1
.	2E	⇒	21D2

Figure 1. Comparing ASCII and Unicode. ASCII, which was designed primarily to represent English, uses 7-bit characters to represent letters, symbols, numbers, and so on. Unicode, which was designed to represent a variety of languages, uses 16-bit characters to represent letters, symbols, and ideographic characters.

it can be considered for standardization. However, it shows great promise as a system that can effectively encode text in many languages.

UNICODE STANDARD

The Unicode[3,4] standard is a fixed-width scheme for encoding written characters, including alphabetic characters, ideographic characters, and symbols. The standard is modeled on the ASCII character set but uses 16-bit encoding to support full multilingual text. Unicode requires no escape sequence or control code to specify a character. Figure 1 illustrates the difference between ASCII text and Unicode text. (All character codes in this article appear in hexadecimal notation.)

The Unicode project began in 1988, and its primary goal is to remedy two serious problems common to most multilingual computer programs:

- They overload the font mechanism when encoding characters.
- They use multiple, inconsistent character codes because of conflicting national and industry character standards.

Major computer companies and organizations incorporated the Unicode Consortium in January 1991 to promote the Unicode standard as an international encoding system for information interchange, to aid in the standard's implementation, and to maintain quality control over future revisions.

Design principles

The Unicode standard's designers envisioned a simple, comprehensive uniform encoding system that could meet the needs of technical and multilingual computing, as well as high-quality typesetting and desktop publishing. The designers tried to achieve a balance between consistency, simplicity, efficiency, and compatibility with existing encoding standards.[3] Unicode's design was based on the following principles:

- *16-bit characters:* All Unicode character codes are 16 bits wide, which provides more code space than the 8-bit ASCII system.
- *Full encoding:* The full 16-bit code space can be used to represent up to 65,536 characters. Unicode Standard Version 2.0 represents 38,885 characters.
- *Encoding of characters, not glyphs:* The Unicode standard encodes characters (the smallest component of a written language that has a semantic value, not glyphs (the shape of a character when it is rendered or displayed).
- *Semantics:* Characters have well-defined semantics. Unicode includes character property tables that can be used with parsing, sorting, and other algorithms that require semantic knowledge about code points.
- *Plain text:* The Unicode standard encodes plain text, rather than fancy (or rich) text, which includes such information as font, size, color, and hypertext links.
- *Logical order:* For all scripts, Unicode text is stored in memory in the same order in which the text is typed, which affects text rendering in some bidirectional scripting systems.
- *Unification:* Because of the limited 16-bit code space, the Unicode standard avoids duplicate encoding of characters. It uses the same code for the same character, regardless of the language that is being represented or the way the language uses the character. So, the letter "a" has the same code whether you are encoding text in English, Spanish, or French.
- *Dynamic composition:* The Unicode standard permits the dynamic composition of accented forms (those letters having diacritical marks). Dynamic composition allows us to form a new character on the fly by combining a character with a diacritical mark.
- *Equivalent sequence:* Some text elements can also be encoded as static precomposed forms—more convenient, compact representations. The Unicode standard provides a way to map to the equivalent dynamic composition, so programs can check for equivalent character sequences.
- *Convertibility:* The standard provides accurate convertibility between Unicode and other widely accepted encoding standards.

Unicode's drawbacks

Although Unicode can remedy some problems experienced by multilingual computer programs, it also has several drawbacks.

Efficiency. Most documents are written entirely in one language, and most languages can be encoded using an 8-bit system. Using a 16-bit encoding scheme uses twice the storage capacity and incurs twice the transmission delay. Although compressing Unicode-encoded files or strings would alleviate this problem, it would have a substantial overhead if done frequently and is thus impractical.

Language orientation. To ensure simple character representation, the Unicode standard is not (and was not) intended to be language-oriented. No information about the language that is being used in a particular piece of text is encoded in a Unicode file.

This can be a problem because in many cases, the same letter exists in different languages. For example, the Latin letter "a" (Unicode 0041) exists in English, Spanish, and French. The Arabic letter "alef" (Unicode 0627) exists in Arabic, Farsi, and Urdu. The same is true for the Chinese letter "zi," Japanese letter "ji," and Korean letter "ja." The same character code (Unicode 5B57) represents these letters.

Different languages use letters or symbols in different ways, and this affects multilingual text processing. You need to know which languages are being used to perform many kinds of character data processing, such as searching, sorting, spell checking, and grammar checking. However, Unicode encoding does not tell you which languages are being used in text files.

A programmer might be able to determine the language of a Unicode text heuristically, based on its content. However, this requires additional work, and the accuracy of such an effort depends on the effectiveness of the heuristic used.

Encoding. The unification principle necessary to Unicode results in a poor encoding of character blocks. For example, the Latin 1 supplement (0080-00FF) mixes over a dozen major European languages. Similarly, the Arabic character block (0600-06FF) encodes characters for Arabic, Persian, Urdu, Pashto, Sindhi, and Kurdish. The same can be said about the unified Chinese, Japanese, and Korean (CJK) ideographs. This causes difficulties in designing applications for a single language, such as those for natural language processing. It is much easier to design such applications if the characters in the underlying block are restricted to one language. This argues for an encoding system that uses separate character blocks for different languages.

Compatibility with ASCII. Unicode is not truly compatible with ASCII. Unicode files are sequences of 16-bit characters, while ASCII files are sequences of 8-bit characters. Unicode's encoding scheme does not recognize ASCII files and requires that you convert them to Unicode format by adding a null byte to the beginning of each ASCII code.

Although conversion from ASCII to Unicode is straightforward, to exchange plain Unicode files over the Internet, you would have to use universal character transformation formats (UTF-7 and UTF-8) with each file.[3,4] It is not clear whether systems that adopt the Unicode standard should convert all ASCII files to Unicode and convert their low-level I/O routines so they can handle Unicode's 16-bit characters.

MULTICODE

In response to the recognized shortcomings of Unicode, I developed Multicode in early 1996.

Multicode's most important feature is its use of multiple character sets. Multicode is not an extension to ASCII but rather a collection of character sets. Most character sets are designed for specific languages, but a few can be designed to be language-independent, representing arrows, mathematical symbols, and so on. Thus, unlike Unicode, Multicode is language-oriented. Each character set is designed to be self-sufficient and, like ASCII, includes all necessary control characters, punctuation, and special symbols.

Character sets

Instead of unifying all languages into a 16-bit character set, Multicode defines 8-bit character sets for most languages and 16-bit character sets for some others, such as Chinese, Japanese, and Korean. There can be a total of 256 character sets, enough to represent all the world's official written languages. (There are about 185 independent countries, and some share the same written language.) Each character set has a unique code; the default set, ASCII, has code 0.

A considerable overlap may exist between two character sets, but each set will have some unique characters and symbols and a unique ordering of characters. For example, Latin languages share many common letters, but each has some unique letters and a unique ordering. The same can be said of Arabic, Farsi, and Urdu, and about East Asian languages. It is also possible to use more than one character set for a language, in cases where a different version of the same language is used in different countries or regions.

Multicode attempts to define a unique character set for each written language, rather than unifying characters across languages as Unicode does. It is also possible to incorporate more than one character set standard in a given language, in case different countries or regions use these sets.

Switching between character sets

Multicode provides a simple and uniform technique for switching between character sets, which is essential for handling text in multiple languages. A *switch character* triggers switching.

For 8-bit character sets, Multicode reserves the last character (code FF) as the switch character. To switch

You need to know which languages are being used to perform many kinds of character data processing, such as searching, sorting, spell checking, and grammar checking.

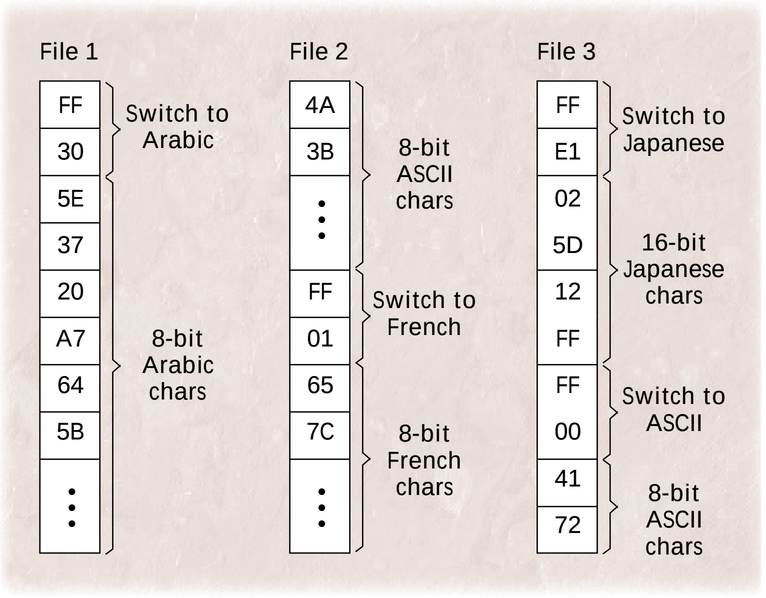


Figure 2. Switching between character sets in Multicode. File 1 is an Arabic text file. To start in Arabic, a switch character (FF) is followed by the code of the Arabic character set (suggested code 30). File 2 mixes ASCII and French characters. Since this file does not start with a switch character, the character set code defaults to 00 (the code for ASCII), an 8-bit character set. Following the ASCII characters is a switch (FF) to French (suggested code 01), another 8-bit character set. File 3 mixes Japanese and ASCII characters. It starts with a switch (FF) to Japanese (suggested code E1), continues with 16-bit Japanese characters, switches to ASCII (FF00), and finishes with 8-bit ASCII characters.

from one 8-bit character set to another 8- or 16-bit set, you would use the switch character and then a second byte, which designates the second character set's code. For example, if you want to switch from French (suggested code 01) to Hindi (code 50) you would use character FF and then 50.

For 16-bit character sets, Multicode reserves the last 256 characters (codes FF00 to FFFF) as switch characters. The first byte of the 2-byte switch character is always FF, while the second byte designates the code of the new character set. Figure 2 shows how to switch between character sets in Multicode. In Files 1 and 2, the switches from the 8-bit character sets were accomplished with two 8-bit characters. In File 3, the switch from 16-bit Japanese was accomplished with one 16-bit character.

Addressing Unicode's drawbacks

Multicode is designed to overcome Unicode's major drawbacks.

Efficiency. Because Multicode uses 8-bit and 16-bit character sets, it can use 8-bit characters for 8-bit documents. It will thus use only half the storage capacity handling such documents that the 16-bit Unicode system would use.

Language orientation. Multicode is designed to be language-oriented. Each character set is designed for a particular language and includes all the necessary control characters and special symbols. Control characters are similar to those used in ASCII, which eliminates the need to switch character sets for control. For example, Multicode incorporates the newline character—a control character at the end of each line in an ASCII file—in the character sets of other languages.

Unlike Unicode, Multicode provides language information directly via the switch characters in Multicode text. This information can be important to language-dependent applications.

Meanwhile, because character sets are language-oriented and small, character and string operations are well-defined and can be implemented easily.

Compatibility with ASCII. Multicode recognizes and is truly compatible with ASCII. You do not even have to use a switch character to begin an ASCII file, since ASCII's character set code (00) is Multicode's default code.

Multicode also recognizes and accommodates the Unicode standard by reserving its last 16-bit character set (FF) for Unicode. In Multicode, you begin a plain Unicode file with the "switch to Unicode" character (FFFF).

Scripting systems

In Multicode, various languages or character sets can share a common *scripting system*. A scripting system designates the rules for rendering characters, including their display, order, and format.[6] Scripting systems differ due to the direction in which the text is written (English is written left-to-right; Arabic, right-to-left), alignment, character representation, and whether a character changes its form depending on its position relative to other characters. One scripting system can serve several languages. For example, the Roman scripting system can serve ASCII, French, German, and other character sets.

A scripting system is responsible for interpreting a character sequence and handling font files. Characters and glyphs can have one-to-one, one-to-many, or many-to-one relationships. The Roman scripting system is relatively simple because character codes have a one-to-one mapping with glyph codes (used to access a glyph in a font file). The Arabic scripting system is complex because it uses all three relationships. For example, an Arabic letter can change its form according to its relative position within a word; such a letter requires a one-to-many mapping. Ligatures, which combine two or more characters into a single glyph, require a many-to-one mapping.

Two characters in different character sets may share the same form (or glyph) but have different codes. For example, the French "a" may be designed to have a dif-

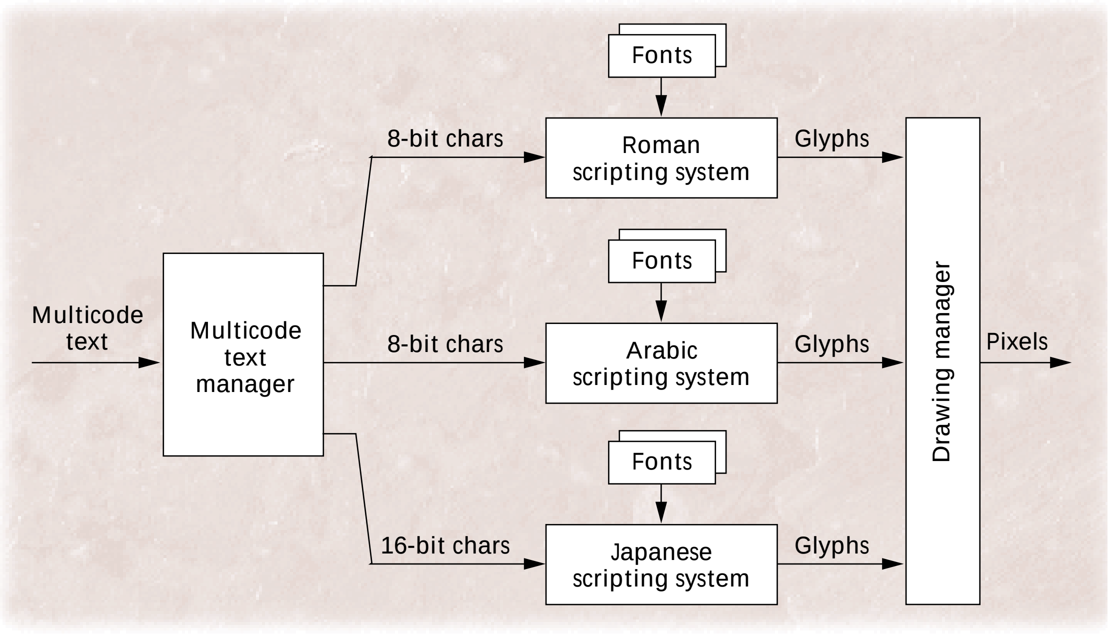


Figure 3. Handling Multicode text. Multicode text is initially received by the Multicode text manager, which directs the text to the scripting system for the language in use. The scripting system converts the character code sequence into a glyph sequence. It then obtains the glyphs from a font file and sends them to the drawing manager for rendering.

ferent character code than the ASCII "a." To render text and handle fonts properly, I suggest the following two alternatives for designing a scripting system.

Fonts corresponding to character sets. A font file could correspond to a unique character set. You would thus have different font files for the ASCII and French character sets. Glyph codes would then have to correspond directly to character codes. Thus, if the character code for the French "a" is different than that of the ASCII "a," the glyph codes should also be different. When switching character sets, the scripting system would have to switch font files as well.

A one-to-one relationship between a font file and character set simplifies the scripting system implementation. It also keeps font files small because they need not include glyphs for characters that are not part of a particular language's character set. However, this approach requires a large number of overlapping font files, particularly for scripting systems that support multiple languages.

Unified fonts. This approach unifies all character sets supported by a scripting system into a character table with no duplicate characters. (The unified character table would be part of the scripting system, so it would not be a concern to application programmers.) Thus, the Latin "a" would represent the ASCII "a," the Spanish "a," the French "a," and so on. You would define a mapping function for each character set supported by a scripting system. This would map a character in a language onto its corresponding unified character. In this case, glyph codes would have to correspond directly to unified characters. Hence, all character sets supported by a scripting system could use the same font file.

This approach would decrease the number of font files. However, each font file would be larger, and a scripting system would have to define mapping functions for each language's character set. This would create a smaller overhead than the previous approach.

Implementing Multicode

Implementing Multicode is modular and incremental. A Multicode-based system need not support all scripting systems simultaneously. Some scripting systems could be added as system extensions implemented in a shared library.

Figure 3 illustrates how a windowing system can handle Multicode text. The Multicode text manager manages all the scripting systems supported by or added to the windowing system. It interprets Multicode text by directing characters to the appropriate scripting system for the language in use. The scripting system converts character codes into glyph codes. In some cases, it may combine several characters into a single glyph code or assign a character a different glyph code based on context. It then obtains the appropriate glyph from a font file and sends the glyph to the drawing manager for rendering.

Both approaches can coexist—Multicode for programming ease and Unicode to support unified fonts.

When the Multicode text manager does not support a language's scripting system, the characters still will not be misinterpreted. When text cannot be rendered or displayed, the Multicode text manager can display a special glyph for every character that cannot be rendered or can display nothing but reports this situation to the user.

PROGRAMMING ISSUES

Because all Unicode characters have a uniform 16-bit width, they can be represented in a programming language by a single character type. In fact, the Unicode standard suggests the type `unichar` to represent Unicode characters. We can define `unichar` as `unsigned short` or as `wchar_t` in C or C++. The Java programming language[5] defines a 16-bit `char` type to represent Unicode characters.

Because of this uniformity, it seems that programming with Unicode should be simpler than programming with Multicode. If displaying characters is the only operation performed on characters, it would indeed be simpler to use a single character type, an advantage in using Unicode. However, it is very difficult to use Unicode for many other types of string operations.

Disadvantages of Unicode

Any programming language's character type is an abstract data type—it represents a set of values and a set of operations on these values. To work consistently, the set of operations would have to be meaningful and uniform across all values of an abstract data type. This would be the case when a character type represents only one language, as is the case with ASCII. However, this would not be the case with Unicode because Unicode's character type is meant to represent many languages, and character and string operations are not meaningful and uniform across languages. Some character and string operations apply to one language but are meaningless to others. Some character and string operations behave differently across languages.

For example, the use of upper or lower case is meaningful in languages that use the Latin alphabet but not in many other languages. Similarly, changing the diacritical marks of letters is meaningful to Arabic but not to ASCII. Meanwhile, some languages that use a similar alphabet order the letters differently. In Unicode, the Arabic letter "ha" (Unicode 0647) comes before the Arabic letter "waw" (Unicode 0648), which is correct in Arabic but not Farsi. Similarly, Swedish treats "ä" (letter "a" with a dieresis, Unicode 00E4) as an individual letter, placing it after "z" in the alphabet. However, German places it either like "ae" or after "a." Unicode string processing cannot handle these problems without information about the language in use.

Programming with Multicode

In Multicode, programs define a string of characters for each character set. This makes sense because character and string operations are language-specific and vary from one character set to another. Thus, instead of having one `char` (or `unichar`) type in a programming language, we would define a family of character types, calling them `ascii`, `french`, `arabic`, `chinese`, and so on. These character types need not be built in or predefined by the programming language. Instead, we can define them in a standard library that supports the programming language.

Strings are arrays of either 8- or 16-bit characters; we cannot mix the two. Because a program defines a string over exactly one character set, a string should not contain a switch character. Data types interpret such characters as invalid values. Thus, it is not possible to mix characters of two data types in one string in a programming language.

Character and string operations. Defining a character data type for exactly one character set simplifies the implementation of these operations. Each character type defines a set of operations meaningful to the corresponding language. We can also define some operations for more than one character type. These operations are overloaded and may behave differently for different character types. It is also possible to design an algorithm for use by several data types. For example, a string comparison algorithm designed for the `ascii` character type should also work without modification for the `french` character type, provided the French character set is properly ordered.

Writing to a file. Writing a character or a string to an output file is straightforward. The programming language I/O library introduces a switch character every time the system encounters a character or string that belongs to a new character set. Clearly, the write procedure should be overloaded to handle many character types.

Reading from a file. Reading characters from a file occurs at two levels. At the lowest level, an operating system treats a Multicode file as a sequence of bytes, and a *read system* call specifies the number of bytes to be read. The operating system associates no meaning to these bytes.

The programming-language level uses a read procedure to read characters. The read procedure is overloaded to handle characters of different types. The type of the actual character parameter should match the character set code of the read characters. Otherwise, no character is read. The program continues reading until it encounters a switch character, but does not read the switch character.

When this happens, the program determines the character set code of the characters that follow. A `charset` function performs this task, returning the character set code of the next character to be read. If the next character is a switch character, `charset` skips that character and returns a new character set code. Otherwise, it returns the current code without changing the file position.

Multicode addresses many of Unicode's drawbacks and should have considerable appeal to programmers who work with text in a variety of languages. Its future, however, depends on the computer industry's acceptance. Multicode can represent Unicode files because it reserves a character set for Unicode. Converting Multicode to Unicode is also straightforward (although the opposite is not). Thus, both approaches can coexist—Multicode for programming ease and Unicode to support unified fonts. ❖

References

1. *ANSI X3.4: Coded Character Set—7-Bit American National Standard Code for Information Interchange*, Am. Nat'l Standards Inst., New York, 1986.
2. *ISO 8859: Information Processing—8-Bit Single-Byte-Coded Graphic Character Sets*, Int'l Organization for Standardization, Geneva, 1987.
3. Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison-Wesley, Reading, Mass., 1996.
4. Unicode Consortium home page, http://www.unicode.org.
5. D. Flanagan, *Java in a Nutshell*, O'Reilly & Associates, Sebastopol, Calif., 1996.
6. Apple Computer, *Inside Macintosh: Text*, Addison-Wesley, Reading, Mass., 1993.

Muhammad F. Mudawwar is an assistant professor in the Computer Science Department at the American University in Cairo. His research interests include parallel programming language design and implementation, multilingual systems, and shared memory multiprocessors. Mudawwar received a BS in electrical engineering from the American University, Beirut, Lebanon, and an MS and a PhD in computer engineering from Syracuse University. He is a member of the IEEE and the ACM.

Contact Mudawwar at mudawwar@acs.auc.eun.eg.

How to Reach *Computer*

Writers
We welcome submissions. For detailed information, write for a Contributors' Guide (computer@computer.org) or visit our Web site: http://computer.org/pubs/computer/computer.htm.

Letters to the Editor
Please provide an e-mail address or daytime phone with your letter.

Computer Letters
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
fax (714) 821-4010
computer@computer.org

On the Web
Visit our Web site at http://computer.org for information about joining and getting involved with the Computer Society and *Computer*.

Magazine Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Make sure to specify *Computer*.

Membership Change of Address
Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies
If you are missing an issue or received a damaged copy, contact membership@computer.org.

Reprints
We sell reprints of articles. For price information or to order, send a query to computer@computer.org or a fax to (714) 821-4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.

COMPUTER
Innovative technology for computer professionals