A Framework for Multilingual Information Processing

by

Steven Edward Atkin

Bachelor of Science
Physics
State University of New York, Stony Brook
1989

Master of Science
in Computer Science
Florida Institute of Technology
1994

A dissertation
submitted to the College of Engineering at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
December, 2001

We the undersigned committee hereby recommend
that the attached document be accepted as fulfilling in
part the requirements for the degree of
Doctor of Philosophy of Computer Science

"A Framework for Multilingual Information Processing,"
a dissertation by Steven Edward Atkin

____
Ryan Stansifer, Ph.D.
Associate Professor, Computer Science
Dissertation Advisor

____
Phil Bernhard, Ph.D.
Associate Professor, Computer Science

____
James Whittaker, Ph.D.
Associate Professor, Computer Science

____
Gary Howell, Ph.D.
Professor, Mathematics

____
William Shoaff, Ph.D.
Associate Professor and Head, Computer Science

Abstract

Title: A Framework for Multilingual Information Processing
Author: Steven Edward Atkin
Major Advisor: Ryan Stansifer, Ph.D.

Recent and (continuing) rapid increases in computing power now enable more of humankind's written communication to be represented as digital data. The most recent and obvious changes in multilingual information processing have been the introduction of larger character sets encompassing more writing systems. Yet the very richness of larger collections of characters has made the interpretation and processing of text more difficult. The many competing motivations (satisfying the needs of linguists, computer scientists, and typographers) for standardizing character sets threaten the purpose of information processing: accurate and facile manipulation of data. Existing character sets are constructed without a consistent strategy or architecture. Complex algorithms and reports are necessary now to understand raw streams of characters representing multilingual text.

We assert that information processing is an architectural problem and not just a character set problem. We analyze several multilingual information processing algorithms (e.g., bidirectional reordering and character normalization) and we conclude that they are more dangerous than beneficial. The countless number of unexpected interactions suggest a lack of a coherent architecture. We introduce abstractions, novel mechanisms, and take the first steps towards organizing them into a new architecture for multilingual information processing. We propose a multilayered architecture which we call Metacode where character sets appear in lower layers and protocols and algorithms in higher layers. We recast bidirectional reordering and character normalization in the Metacode framework.

Table of Contents

List of Figures ix

List of Tables xi

Acknowledgement xiii

Dedication xiv

Chapter 1 Introduction 1

1.1 Architecture Overview 3
1.2 Problem Statement 5
1.3 Outline of Dissertation 5

Chapter 2 Software Globalization 7

2.1 Overview 7
2.2 Translation 9
2.3 International User Interfaces 10
2.3.1 Metaphors 10
2.3.2 Geometry 11
2.3.3 Color 11
2.3.4 Icons 12
2.3.5 Sound 12
2.4 Cultural and Linguistic Formatting 13
2.4.1 Numeric Formatting 13
2.4.2 Date and Time Formatting 14
2.4.3 Calendar Systems 14
2.4.4 Measurement 15
2.4.5 Collating 15
2.4.6 Character Classification 16
2.4.7 Locales 17
2.5 Keyboard Input 17
2.6 Fonts 18
2.7 Character Coding Systems 20

Chapter 3 Character Coding Systems 22

3.1 Terms 22
3.2 Character Encoding Schemes 24
3.3 European Encodings 24

3.3.1 ISO 7-bit Character Encodings . . . 27
3.3.2 ISO 8-bit Character Encodings . . . 29
3.3.3 Vendor Specific Character Encodings . . . 33
3.4 Japanese Encodings . . . 40
3.4.1 Personal Computer Encoding Method . . . 43
3.4.2 Extended Unix Code Encoding Method . . . 45
3.4.3 Host Encoding Method . . . 46
3.4.4 Internet Exchange Method . . . 47
3.4.5 Vendor Specific Encodings . . . 49
3.5 Chinese Encodings . . . 50
3.5.1 Peoples Republic of China . . . 51
3.5.2 Republic of China . . . 53
3.5.3 Hong Kong Special Administrative Region . . . 54
3.6 Korean Encodings . . . 55
3.6.1 South Korea . . . 55
3.6.2 North Korea . . . 55
3.7 Vietnamese Encodings . . . 56
3.8 Multilingual Encodings . . . 57
3.8.1 Why Are Multilingual Encodings Necessary? . . . 57
3.8.2 Unicode and ISO-10646 . . . 59
3.8.2.1 History of Unicode . . . 60
3.8.2.2 Goals of Unicode . . . 60
3.8.2.3 Unicode's Principles . . . 61
3.8.2.4 Differences Between Unicode and ISO-10646 . . . 62
3.8.2.5 Unicode's Organization . . . 63
3.8.2.6 Unicode Transmission Forms . . . 67
3.8.3 Criticism of Unicode . . . 68
3.8.3.1 Problems With Character/Glyph Separation . . . 69
3.8.3.2 Problems With Han Unification . . . 69
3.8.3.3 ISO-8859-1 Compatibility . . . 70
3.8.3.4 Efficiency . . . 71
3.8.4 Mudawwar's Multicode . . . 71
3.8.4.1 Character Sets in Multicode . . . 71
3.8.4.2 Character Set Switching in Multicode . . . 72
3.8.4.3 Focus on Written Languages . . . 73
3.8.4.4 ASCII/Unicode Compatibility . . . 74
3.8.4.5 Glyph Association in Multicode . . . 74
3.8.5 TRON . . . 74
3.8.5.1 TRON Single Byte Character Code . . . 75
3.8.5.2 TRON Double Byte Character Code . . . 76
3.8.5.3 Japanese TRON Code . . . 76
3.8.6 EPICIST . . . 77

3.8.6.1 EPICIST Code Points 78
3.8.6.2 EPICIST Character Code Space 78
3.8.6.3 Compatibility With Unicode 78
3.8.6.4 Epic Virtual Machine 79
3.8.6.5 Using the Epic Virtual Machine for Ancient Symbols 79
3.8.7 Current Direction of Multilingual Encodings 80

Chapter 4 Bidirectional Text 81

4.1 Non Latin Scripts 82
4.1.1 Arabic and Hebrew Scripts 83
4.1.1.1 Cursive 83
4.1.1.2 Position 84
4.1.1.3 Ligatures 84
4.1.1.4 Mirroring 85
4.1.2 Mongolian Script 85
4.2 Bidirectional Layout 86
4.2.1 Logical and Display Order 86
4.2.2 Contextual Problems 87
4.2.3 Domain Names 88
4.2.4 External Interactions 89
4.2.4.1 Line Breaking 89
4.2.4.2 Glyph Mapping 89
4.2.4.3 Behavioral Overrides 90
4.2.5 Bidirectional Editing 90
4.2.6 Goals 90
4.3 General Solutions to Bidirectional Layout 91
4.3.1 Forced Display 91
4.3.2 Explicit 91
4.3.3 Implicit 92
4.3.4 Implicit/Explicit 92
4.4 Implicit/Explicit Bidirectional Algorithms 92
4.4.1 Unicode Bidirectional Algorithm 93
4.4.2 IBM Classes for Unicode (ICU) and Java 93
4.4.3 Pretty Good Bidirectional Algorithm (PGBA) 94
4.4.4 Free Implementation of the Bidirectional Algorithm (FriBidi) 94
4.5 Evaluation of Bidirectional Layout Algorithms 94
4.5.1 Testing Convention 95
4.5.2 Test Cases 96
4.5.3 Test Results 100
4.6 Functional Approach to Bidirectional Layout 102
4.6.1 Haskell Bidirectional Algorithm (HaBi) 103
4.6.1.1 HaBi Source Code 106

4.6.1.2 Benefits of HaBi . . . 107
4.7 Problems With Bidirectional Layout Algorithms . . . 108
4.7.1 Unicode Bidirectional Algorithm . . . 109
4.7.2 Reference Implementation . . . 110
4.7.3 HaBi . . . 110
4.8 Limitations of Strategies . . . 111
4.8.1 Metadata . . . 111

Chapter 5 Enhancing Plain Text . . . 112

5.1 Metadata . . . 113
5.1.1 Historical Perspective . . . 113
5.2 Unicode Character Model . . . 116
5.2.1 Transmission Layer . . . 117
5.2.2 Code Point Layer . . . 117
5.2.3 Character/Control Layer . . . 119
5.2.4 Character Property Layer . . . 120
5.3 Strategies for Capturing Semantics . . . 121
5.3.1 XML . . . 121
5.3.2 Language Tagging . . . 123
5.3.2.1 Directional Properties of Language Tags . . . 125
5.3.3 General Unicode Metadata . . . 126
5.4 Encoding XML . . . 130
5.5 New XML . . . 134
5.6 Text Element . . . 134
5.7 Metadata and Bidirectional Inferencing . . . 138
5.7.0.1 HTML and Bidirectional Tags . . . 142
5.8 New Architecture . . . 143

Chapter 6 Metacode . . . 144

6.1 Metacode Architecture . . . 144
6.2 Metacode Compared to Unicode . . . 147
6.2.1 Transmission Layer . . . 147
6.2.2 Code Point Layer . . . 147
6.2.3 Character Layer . . . 148
6.2.3.1 Combining Characters . . . 148
6.2.3.2 Glyph Variants . . . 152
6.2.3.3 Control Codes . . . 153
6.2.3.4 Metadata Tag Characters . . . 155
6.2.4 Character Property Layer . . . 155
6.2.5 Tag Definition Layer . . . 157
6.2.6 Metacode Conversion . . . 158

6.2.7 Content Layer . . . 162
6.3 Data Equivalence . . . 162
6.3.1 Unicode Normalization . . . 162
6.3.1.1 Unicode Normal Forms . . . 163
6.3.1.2 Unicode Normalization Algorithm . . . 166
6.3.1.3 Problems with Unicode Normalization . . . 168
6.3.2 Data Equivalence in Metacode . . . 171
6.3.3 Simulating Unicode in Metacode . . . 174
6.4 Code Points vs. Metadata . . . 175
6.4.1 Metacode Principles . . . 175
6.4.2 Applying Metacode Heuristics . . . 176
6.4.2.1 Natural language text . . . 176
6.4.2.2 Mathematics . . . 177
6.4.2.3 Dance notation . . . 178
6.5 Benefits of Metacode . . . 180

Chapter 7 Conclusions . . . 182

7.1 Summary . . . 182
7.2 Contributions . . . 184
7.3 Limitations . . . 188
7.4 Future Work . . . 188

References . . . 190

Appendix A . . . 201

Appendix B . . . 205

Appendix C . . . 206

Appendix D . . . 209

List of Figures

Figure 2-1. Using Metaphors 11
Figure 2-2. Macintosh trash can icon 12
Figure 2-3. British post box 12
Figure 2-4. French in France and French in Canada 16
Figure 2-5. Character to glyph mapping 19
Figure 2-6. Contextual glyphs 19
Figure 2-7. Japanese furigana characters 19
Figure 3-1. ASCII encoding 25
Figure 3-2. ISO-8859-1 encoding 31
Figure 3-3. ISO-8859-7 encoding 32
Figure 3-4. EBCDIC encoding 35
Figure 3-5. IBM 850 37
Figure 3-6. Windows 1252 39
Figure 3-7. Japanese Kanji characters 40
Figure 3-8. Japanese Katakana characters 41
Figure 3-9. Japanese Hiragana characters 41
Figure 3-10. Mixed DBCS and SBCS characters 44
Figure 3-11. JIS X0212 PC encoding 45
Figure 3-12. EUC encoding 46
Figure 3-13. ISO-2022 encoding 48
Figure 3-14. ISO-2022-JP encoding 49
Figure 3-15. Forms of UCS-4 and UCS-2 63
Figure 3-16. Unicode layout 65
Figure 3-17. Surrogate conversion 66
Figure 3-18. Character set switching in Multicode 73
Figure 3-19. TRON single byte character code 75
Figure 3-20. TRON double byte character code 77
Figure 4-1. Tunisian newspaper 81
Figure 4-2. Ambiguous layout 87
Figure 4-3. Rendering numbers 87
Figure 4-4. Using a hyphen minus in a domain name 88
Figure 4-5. Using a full-stop in a domain name 89
Figure 4-6. Input and output of Haskell Bidirectional Reference 104
Figure 4-7. Data flow 105
Figure 5-1. Using LTRS and FIGS in Baudot code 114
Figure 5-2. ISO-2022 escape sequences 116
Figure 5-3. Unicode Character Model 117
Figure 5-4. Compatibility Normalization 123
Figure 5-5. Language tag 124
Figure 5-6. Error in bidirectional processing 126
Figure 5-7. Error in language tagging 126
Figure 5-8. Regular expressions for tags 129

Figure 5-9. Regular expression for text stream ... 129
Figure 5-10. Sample tag ... 130
Figure 5-11. Alternative language tag ... 130
Figure 5-12. Sample XML code ... 131
Figure 5-13. Sample XML code encoded in metadata ... 133
Figure 5-14. Combining characters ... 135
Figure 5-15. Joiners ... 137
Figure 5-16. ELM tag ... 137
Figure 5-17. Mapping from display order to logical order ... 139
Figure 5-18. Example output stream ... 140
Figure 5-19. Mathematical expression ... 141
Figure 5-20. BDO tag syntax ... 142
Figure 5-21. Using HTML bidirectional tags ... 142
Figure 6-1. New Text Framework ... 145
Figure 6-2. Combining character protocol ... 149
Figure 6-3. Ligature protocol ... 153
Figure 6-4. Spacing protocol ... 155
Figure 6-5. Interwoven protocols ... 158
Figure 6-6. Metacode code point protocol ... 159
Figure 6-7. ISO-2022 escape sequence in Metacode ... 161
Figure 6-8. Non interacting diacritics ... 163
Figure 6-9. Compatibility equivalence ... 163
Figure 6-10. Conversion to NFKD ... 167
Figure 6-11. Conversion to NFD ... 168
Figure 6-12. Protocol interaction ... 169
Figure 6-13. Data mangling ... 170
Figure 6-14. Question Exclamation Mark ... 174
Figure 6-15. Metadata Question Exclamation Mark ... 174
Figure 6-16. Simulating Unicode normalization ... 175
Figure 6-17. Egyptian hieroglyphic phonogram ... 177
Figure 6-18. Egyptian hieroglyphic ideograph ... 177
Figure 6-19. Mathematical characters ... 178
Figure 6-20. Action Stroke Dance Notation ... 179
Figure 6-21. Action Stroke Dance Notation with movement ... 180
Figure 6-22. Metacode Action Stroke Dance Notation tag ... 180

List of Tables

Table 3-1. ASCII control codes 25
Table 3-2. ISO variant characters 28
Table 3-3. French version of ISO-646 28
Table 3-4. ISO-8859 layout 30
Table 3-5. ISO-8859 standards 33
Table 3-6. EBCDIC layout 34
Table 3-7. IBM PC code pages 36
Table 3-8. Windows code pages 38
Table 3-9. JIS character standards 43
Table 3-10. ISO-2022-JP escape sequences 49
Table 3-11. Vendor encodings 49
Table 3-12. GB standards 53
Table 3-13. Taiwanese standards 54
Table 3-14. Unicode code point sections 64
Table 3-15. UTF-8 68
Table 3-16. Unicode transformation formats 68
Table 4-1. Bidirectional character mappings for testing 95
Table 4-2. Arabic charmap tests 96
Table 4-3. Hebrew charmap tests 97
Table 4-4. Mixed charmap tests 98
Table 4-5. Explicit override tests 99
Table 4-6. Arabic test differences 100
Table 4-7. Hebrew test differences 101
Table 4-8. Mixed test differences 101
Table 5-1. Problem Characters 119
Table 5-2. Character Properties 120
Table 5-3. Tag characters 128
Table 5-4. Other text element tags 138
Table 6-1. Excluded Unicode combining characters 150
Table 6-2. Redefined Unicode combining characters 150
Table 6-3. Unicode glyph composites 153
Table 6-4. Unicode spacing characters 154
Table 6-5. Metacode tag characters 155
Table 6-6. Metacode character properties 156
Table 6-7. Metacode case property values 156
Table 6-8. Metacode script direction property values 156
Table 6-9. Metacode code point value property 156
Table 6-10. Metacode tag property values 156
Table 6-11. Major protocols 158
Table 6-12. Converting deprecated Unicode code points to Metacode 159
Table 6-13. Normalization forms 164
Table 6-14. The string “flambé” 166

Table 6-15. The string “flambé” in Metacode 173

Acknowledgement

There is one person above all others who deserves my sincerest thanks and respect for his continuous support during the writing of this dissertation: my advisor, Dr. Ryan Stansifer. I could not have completed it without him.

There are many other people who contributed to this dissertation in many ways. First, I would like to thank my colleague, Mr. Ray Venditti, for securing my funding at IBM. He made the impossible possible. Second, my committee, Dr. Phil Bernhard, Dr. James Whittaker, and Dr. Gary Howell. They fostered a stress-free working relationship which was critical to the completion of this dissertation. Third, my uncle, Dr. Jeffrey Title for all the hours of consultation, comments, and late nights. Lastly, Dr. Phil Chan for his invaluable suggestions.

Dedication

To my wife Sevim

To my Parents

To my Aunt and Uncle

1 Introduction

Computers process information in many natural languages. We call multilingual information processing the manipulation and storage of natural language text. We exclude symbolic, numeric, and scientific information processing. Natural language text is composed of characters. We employ discrete and finite sets of characters in computers to capture text. Assigning different kinds of characters yields different kinds of text. Current information processing methods, cobbled together over time, capture natural language text imperfectly in information systems due to an over reliance on rigid character sets (such as, ASCII and EBCDIC). This dissertation is directed towards identifying and correcting existing challenges and imperfections in multilingual information processing.

The need for extremely accurate and elegant multilingual information processing has become pressing as the preponderance of data processing tasks change from almost exclusively numerical to increasingly text oriented. The world community increasingly depends on written natural language text. E-mail has become a standard for messaging within and between companies and governments and among millions (perhaps billions) of individuals throughout the world.

For the most part, the information stored in computers codifies mankind's writing (as opposed to other communication like art, music, speech, etc.) In this dissertation we are concerned exclusively with text — the text that represents the writing systems of the world. Because natural language writing systems are diverse we have information processing challenges and limitations. We are not likely to soon solve all the difficulties presented. This dissertation will address the challenges and

limitations inherent in the diverse and complex multilingual environment. We propose a comprehensive plan for fundamental improvements in the conceptualization and implementation of a more effective approach to multilingual information processing.

Multilingual information processing, to date, has largely been centered around expanding and consolidating character sets. This character set centric approach enables software developers to merge text files comprised from disparate character sets, hopefully reducing the complexity and cost of text processing. This oversimplified strategy has added significant problems for the designers of multilingual software.

In the character set centric approach text processing is viewed as a display oriented activity. This has resulted in the creation of character sets which maintain distinctions only for the sake of appearance or convenience of presentation. This bias towards display makes it more difficult to use character data for purposes other than presentation. Digital information becomes hidden or lost without a clear and unique choice for representing multilingual text.

In this dissertation we demonstrate that the character set centric perspective is restrictive, myopic, inconsistent, and clumsy. Character sets play a role, but cannot by themselves solve multilingual information processing problems. A subtle paradigm shift which we delineate in this dissertation allows a more natural solution to information processing. In our view information processing is best approached as an architectural problem. We introduce abstractions, general algorithms, novel mechanisms, and take the first steps towards organizing them into a new architecture for multilingual information processing.

In our architecture roles and responsibilities are clearly separated, fundamental abstractions are insulated from change. This separation goes to the heart of our approach. By using the correct abstractions the common tasks only need to be written

once — correctly and cleanly. Additional applications can be built from this well defined set and achieve more functionality. Writing systems that have been given little attention can be accommodated without harmful interactions. Improvements of fidelity can happen without endless tinkering. It has been and will continue to be a continuous struggle to capture language more perfectly, even English. It is the nature of living languages to grow and evolve new forms. Change will inevitably happen. With the appropriate abstractions we minimize the potential trouble of adapting to these changes.

Higher-level mechanisms, such as XML-like technologies, offer a solution to some of the problems we attack. In several cases we show how to make use of these mechanisms to solve multilingual information processing problems. We show that there are limitations, however. Higher-level (XML-like protocols) and lower-level (character sets) mechanisms often attack the same problem independently and from different directions. This leads to redundant, overlapping, conflicting, and ill-suited solutions. These mechanisms are twisted into roles for which they are not suited. This strategy is an uncoordinated attack on a general problem. Our architecture provides a clearer division of roles and responsibilities.

We acknowledge that it is not easy to switch directions in the development of multilingual information processing. We believe that there is a practical migration path and from time to time we point out how this can be facilitated. In fact, the changes we recommend to existing standards to accomplish our new approach are relatively small.

1.1 Architecture Overview

In this dissertation we decompose multilingual information processing into its main components. We approach the design of an architecture for multilingual information processing from a ground-up strategy. In our architecture we develop stackable layers which are similar to the layers found in networking architectures.

Each layer builds upon the abstraction of the prior lower layer. The various aspects common to writing systems are dissected and layered. These layers are used to construct an architecture for multilingual information processing. The number of architectural layers is large enough to allow for a clean separation of responsibilities, but not so small as to group unrelated functions together out of necessity. Each layer in our architecture serves a well defined purpose with clear and simple interfaces. This dissertation describes these architectural layers and establishes relationships between them.

In our framework we incorporate character sets, protocols and algorithms. Just like the character set centric view, we also use characters for representing text. In the character set centric view great emphasis is placed on viewing and printing text. This has resulted in the creation of character sets maintaining distinctions that are founded not in meaning, but only in appearance, ultimately leading to solutions that favor presentation, rather than content. In contrast to the character set centric view characters in our architecture are distinguished by their underlying meaning.

We acknowledge that at times it may be necessary and useful to maintain distinctions based on appearance. Our architecture allows a focus on content without sacrificing display. However, we depart from the usual mechanisms of the character set centric approach and rely instead on higher-order protocols to capture this information.

In some cases information processing algorithms require information over and above the individual characters, for example; language information, line breaking, and non-breaking spaces. In the character set centric model such control information is coerced into characters. In many cases this has occurred in an ad-hoc haphazard way leading to character sets with confusing semantics. In our architecture we fix this problem through the use of well defined protocols to capture more of the underlying structure of text than is possible with character sets alone.

In this dissertation we are not specifically concerned with the definition of text protocols, rather we aim to develop a flexible, open, and extensible mechanism for their definition. We propose a multi-layered architecture where character sets appear in lower layers, and protocols and algorithms are in the higher layers. Our architecture is flexible — it is unnecessary to make character set changes as new protocols are adopted. The architecture is open — there is no limit to the number of possible protocols. Protocols in our architecture are extensible — numerous protocols can be interwoven without ill-effect.

In the character set centric model general purpose information processing algorithms are difficult to write, because of confusing character semantics and the overall bias towards display. Our architecture has no such bias and confusing semantics. Therefore, it is easier to construct general purpose information processing algorithms. In our architecture we provide a core set of general purpose algorithms that we believe are crucial for multilingual information processing.

1.2 Problem Statement

We seek to define and organize the primary components for multilingual information processing that unambiguously separates content, display and control information.

1.3 Outline of Dissertation

Chapter 2 describes the overall field of software globalization. We introduce the concepts of internationalization, localization, and translation. We concentrate on the problems encountered during the creation of multilingual software. In particular, we look at user interface, cultural formatting, keyboard input, and character set problems.

Chapter 3 presents an in depth analysis of character sets and character coding systems. We start the chapter by introducing and defining the relevant terms related

to character coding systems. We describe in detail several monolingual character character sets that cover both ideographic and syllabic scripts. We conclude the chapter with an examination of multilingual character coding systems.

Chapter 4 considers multilingual character coding problems. We examine the trouble caused when Arabic and English text are mixed in an information processing environment. This mixed text is called bidirectional text — text with characters written left-to-right and right-to-left. We examine several strategies for processing and displaying bidirectional text. We find the existing strategies inadequate, presenting evidence that the underlying character set centric model is insufficient.

Chapter 5 explores several strategies for addressing the shortcomings of the character set centric model. In particular, we look at using metadata (data describing data) to describe more of the underlying structure of scripts. We look at XML as a metadata model for multilingual information processing, but find it unsuitable. We define our own general metadata model, presenting evidence of its suitability for multilingual information processing. In demonstrating the features of our metadata model we return to the bidirectional text problem. We find that the general metadata model allows for a general reorganization of multilingual information processing.

Chapter 6 introduces our multilingual information processing architecture. Our architecture incorporates character sets, metadata, and core protocols, providing an overall framework for multilingual information processing. We call this architecture Metacode. To demonstrate the features of our architecture we use several writing systems as examples. We conclude by summarizing the benefits our architecture provides.

Chapter 7 discusses our contributions, limitations, and future work.

2 Software Globalization

Software globalization is concerned with the application of practices and processes to make a software product usable throughout the world. The term *globalization* refers to the whole process starting from an internationalized version of an application through the production of multiple localized versions of it. The three terms *internationalization*, *localization*, and *translation* broadly define the various subdisciplines of software globalization. Software globalization builds upon *internationalization*, *localization,* and *translation*. In this dissertation we define *internationalization* as the process of creating cultural and language neutral software. The term *localization* refers to the process of adapting a software product to a specific culture and language. The term *translation* is defined as the process of converting human readable text from one language into another. [55]

Throughout this chapter we discuss the primary subdisciplines that make up the field of software globalization. In each subdiscipline we outline the principle issues and current trends. The software globalization area is a relatively new field of study, and as we will see the boundaries of the field are still open to debate.

2.1 Overview

Unlike, other research areas of computer science, software globalization has arisen not from academia, but rather from industry. We argue that this industrial movement has occurred for the following reasons: increasing user expectations, proliferation of distributed computing, explosive software development costs, compounding maintenance outlays, and governmental requirements.

The market for computing in the 1950's and early 1960's was not home users, but rather large institutions (banking, industrial, governmental, and research). In many cases computing at these large institutions was primarily for the purposes of number crunching. Therefore, linguistic/cultural support was not a strong requirement. In fact, even English was not fully supported (lower case characters do not appear until 1967). Moreover, if a system did provide additional linguistic support it was usually a language that could be represented using the Latin script.

In the late 1970's we see a dramatic shift in computing from number crunching to information processing. This shift, coupled with the advent of personal computing in the 1980's, caused user expectations to rise. It was no longer possible to insist that users conform to the computer/software, but rather software must adapt to the user, making linguistic/cultural support a must. Most commercial software during this time period was developed within the United States, so development teams had little experience with linguistic/cultural issues. Support for other languages/cultures was viewed as customization/translation and not as part of main line development. This customization/translation activity was farmed out to another organization within the company or an outside localization firm. Therefore, software development/delivery became staged. By staged we mean, first the US version would be delivered followed by various language versions. This overall strategy caused software development costs to explode. [29],[61],[85]

The ever increasing costs can be attributed to two factors. First, customization, in reality, requires more than just translation, because in many cases the source code itself required modification. In some cases source modifications were incompatible with changes made for other languages. It became necessary to maintain separate source lines for the individual languages, further increasing development costs. Maintaining these separate source lines had a rippling effect throughout the entire software development process. Its effects were most strongly felt during maintenance, because in many cases corrective fixes could not be applied universally across

the various language versions. Multiple fixes would be created, further increasing the overall software development cost. [61],[85]

The multiple source line development strategy had a direct impact upon customers, in particular large multinational customers. Most multinational customers wanted the ability to simultaneously roll out multiple language software solutions across their entire corporation. This was difficult to achieve, because of the staged development/delivery method in effect at the time. Each specific language version was based upon a different source line, thereby requiring the customer to repeat the entire certification process for each language. By the late 1980's this strategy became crippling as costs sky rocketed. [70]

A strong push from multiple source line development to single source line development was undertaken in the 1990's. It is at this point in time that we see the beginnings of internationalization and localization. It was a move in the right direction, but the internationalization problem turned out to involve a lot more factors than simply translating and converging multiple source lines. In the sections below we explore these factors, which in turn form the six individual sub fields of software globalization: Translation, International User Interfaces, Cultural/Linguistic Formatting, Keyboard Input, Fonts, and Character Coding Systems.

2.2 Translation

Naturally, a software system must be translated into the user's native language for it to be useful. At first it might seem that translating a programs message's and user interface elements is a relatively simple task. There are subtleties, however.

Literal translations of text strings in applications without regard for human factors principles may serve as a source for confusing or misleading user interfaces. One such example comes form the Danish translation of MacPaint. One of the menus in MacPaint is called "Goodies" which is acceptable as the name of a menu in

English. The literal translation of goodies in Danish is the word godter. This is a proper translation, but has an entirely different connotation (mostly having to do with candy), leading users to confusion. [8]

Translation of user manuals must also be approached with caution. For example, the following is a sample of an English translation from the preface of a printer manual in Japanese:

> "We sincerely expect that the PRINTER CI-600 will be appreciated more than ever, in the fields of 'data-transformation' by means of human-scale, and the subsequent result of 'fluent metabolism' as regards the artificial mammoth creature-systematized information within the up-to-date human society."

Clearly this example demonstrates that the problem of translation is one of global importance. [8]

2.3 International User Interfaces

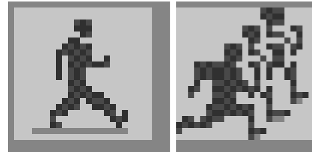
Software must fit into the cultural context of the user. Internationalized applications should appear as if they were custom designed for each individual market. This requires a deep understanding of not only languages, but also of cultural norms, taboos, and metaphors. [91],[8]

2.3.1 Metaphors

Just as metaphors permeate our everyday communication, so do they occur throughout the interfaces we use. It is the use of metaphors that makes our interaction with software seem more natural and intuitive. For example, the icons shown in Figure 2-1. are from Lotus 1-2-3. The second icon, with somebody running, is intended to let you "run" a macro [29]. The problem is that not everyone "runs" a macro. In France, for example, the metaphor isn't running a macro it's "throwing" a macro. So using the metaphor of somebody running makes no sense for users in France. What may seem natural to users in the United States may not appear to be

obvious to users in other markets. Therefore, as software is internationalized and localized it is imperative that metaphors be tailored for each market just as language is. [25],[8]

Figure 2-1. Using Metaphors



2.3.2 Geometry

Even visual scanning patterns are culturally dependent. Studies with English and Hebrew readers have demonstrated the existence of these differences. English readers tend to start scanning an object from the left quadrant, while Hebrew readers tend to scan from the right quadrant. This is probably due to the fact that each language has a specific scanning direction (English left-to-right and Hebrew right-to-left). This infers that the formatting and positioning of windows in an application must also be tailored for each market. [1],[8]

2.3.3 Color

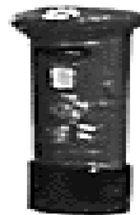
The varied use of color in everyday things is common. However, cultural differences can affect the meanings attributed to color. For example, in the United States the color red is often used to indicate danger, while in China the color red represents happiness. The color white in the United States represents hope and purity, however in Japan the color white represents death. By using color correctly, interfaces can be created where color can impart and reinforce information conveyed by other media, such as text. [83],[91],[8]

2.3.4 Icons

The abundant use of icons in today's graphical interfaces aims to provide a more intuitive user interface by equating an underlying action or function to a symbol. The seemingly simple trash icon in the Macintosh interface seems at first to have only one underlying meaning, disposing of files. See Figure 2-2. Nevertheless, in the United Kingdom an entirely different meaning was ascribed to this icon. It turned out that the trash can icon resembled a British post box better than a British trash can, creating confusion. See Figure 2-3. Therefore, even the style of icons is culturally dependent. [37],[38],[81],[91],[8]

Figure 2-2. Macintosh trash can icon

Figure 2-3. British post box



2.3.5 Sound

Not only are icons culturally sensitive, but the use of sound is as well. The use of sound has improved the "user-friendliness" of applications by reinforcing visually displayed messages. At first appearance this seems to be culturally neutral, but Lotus corporation discovered that this was not true. Lotus spent considerable time and money to internationalize 1-2-3 for the Japanese market, but were dismayed to find that 1-2-3 did not receive a favorable reception. Lotus was unaware that the simple "error beep" present in 1-2-3 was a source of great discomfort for the Japanese. The

Japanese tend to work in highly crowded offices and are fearful of letting coworkers know they are making mistakes. Hence it was only natural that the Japanese avoided using 1-2-3 when they were at work. [91],[14]

The range of issues just described broadly defines the area of interest to the international user interface community. The design of international user interfaces is mostly viewed as a facet of localization. This is not a strict rule as there is some amount of overlap with some of the other sub fields of globalization.

2.4 Cultural and Linguistic Formatting

Differences in notational conventions provide yet another area that must undergo internationalization. For example, although all countries have some form of a monetary system, few countries agree on details such as the currency symbol and the formatting of currency. Similarly, most people agree that time is measured by the cycles of the earth and the moon, but the ubiquitous Gregorian calendar used in Europe and North America isn't necessarily utilized all around the world. Such differing views require highly flexible software that can process and represent data for a wide variety of cultures and languages. In the following paragraphs we discuss some of the common issues surrounding cultural and linguistic formatting. [85]

2.4.1 Numeric Formatting

Most regions of the world adhere to Arabic numbers, where values are base 10. Problems arise when large and fractional numbers are used. In the United States, the "radix" point (the character which separates the whole part of the number from the fractional part) is the full-stop. Throughout most of Europe the radix point is a comma. In the United States the comma is used as break for separating groups of numbers. Europeans use single quotes to separate groups of numbers. In addition to the problem of formatting numeric quantities, the interpretation of the values of numbers is also culturally sensitive. For example, in the United States a "billion" is

represented as a thousand million (1,000,000,000), but in Europe a billion is a million million (1,000,000,000,000), which is a substantially larger quantity. [91],[8]

2.4.2 Date and Time Formatting

As numbers are culturally dependent so are dates. For example the date 11/1/1993 is interpreted as November 1st, 1993 in the United States, but throughout most of Europe the date is interpreted as January 11th, 1993. On the surface this does not seem to be a difficult problem to handle, but countries that use non-western calendars (Japan, China, Israel, etc.) all have different ways of keeping track of the date. Just like dates the expression of time is also culturally dependent. For example, in the United States time is based on a 12 hour time table, while other countries prefer a 24 hour system. [91],[8]

2.4.3 Calendar Systems

Many countries use the 12 month, 365 day Gregorian calendar and define January 1 as the first day of a new year. Many Islamic countries (e.g., Saudi Arabia and Egypt) also use a calendar with 12 months, but only 354 or 355 days. Because this year is shorter than the Gregorian year, the first day of the Islamic year changes from year to year on the Gregorian calendar. Therefore, the Islamic calendar has no connection to the physical seasons. [75]

Israelis use the Hebrew calendar that has either 12 or 13 months, based upon whether it is a leap year. Non leap years have 12 months and between 353 and 355 days. In a leap year, however, an extra thirteenth month is added. This extra month allows the Hebrew calendar to stay synchronized with the seasons of the year. [75]

Determining the year also depends on the calendar system. For example, in Japan two calendar systems are used Gregorian and Imperial. In the Imperial system the year is based on the number of years the current emperor has been reigning. When the emperor dies and a new emperor ascends the throne, a new era begins, and the

year count for that era begins. Furthermore, in Japan it is culturally unacceptable to create a software calendar system that allows a user to reset the year to the beginning, since this implies the imminent demise of the current emperor. [75],[14]

2.4.4 Measurement

Systems of measurement also vary throughout the world. For example, most of Europe uses units of measurement that are based upon the MKS system (meters, kilograms, and seconds), while in the United States measurement is based upon SI units (inches, pounds, and seconds). Such differences need to be accounted for within software systems. For example, a word processor should display its ruler in inches when used in the United States, but should display its units in centimeters when used in Europe. [91],[8]

2.4.5 Collating

Naturally, numeric data is not the only kind of data subject to cultural and linguistic influences. The ordering of words (character data) is dependent upon both language and culture. Even in languages that use the same script system, collation orders vary. In Spanish the character, sequence “cho” comes after “co”, because “ch” is collated as a single character, while in English “cho” comes before “co”.

In many cases the sorting conventions used by Asian languages are more complex than the sorting conventions of western languages. In Asian languages ideographic characters occur more frequently than alphabetic characters. The concepts used in sorting alphabetic scripts are not necessarily applicable to ideographic scripts. Ideographic characters represent concepts and thus can be sorted in a variety of ways (e.g., by phonetic representation, by the character’s base building block, or by the number of strokes used to write the character).

2.4.6 Character Classification

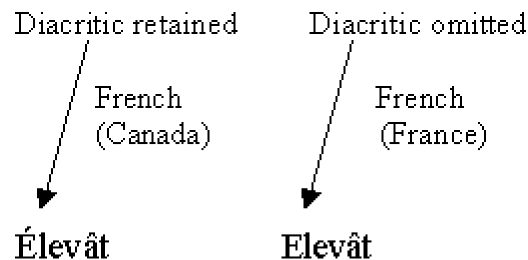
In the development of internationalized software we often find it useful to classify characters (e.g., alphabetic, uppercase, lowercase, numeric, etc.) because it eases processing of character data. Different languages, however, may classify characters differently. For example, in English only 52 characters are classified as alphabetic. Danish classifies 112 characters as alphabetic. [85]

Many languages have no concept of case (lowercase and uppercase). Languages based on Ideographic characters (e.g., Chinese, Japanese, and Korean) all have a single case. This single case property can also be found in some phonetic scripts (e.g., Arabic and Hebrew). [75]

Even languages that make case distinctions do not always use the same case conversion rules. For example, in German the Eszett character "ß" is a lowercase character that gets converted to two uppercase S's. Therefore, not every lowercase character has a simple uppercase equivalent. [75]

In some situations even the same language may use different case conversion rules. For example, in French when diacritic characters are converted from lowercase to uppercase the diacritics are usually removed. On the other hand, in French Canadian diacritics are retained during case conversion. See Figure 2-4. Therefore, knowing only the language does not guarantee proper case conversion. [75]

Figure 2-4. French in France and French in Canada



2.4.7 Locales

The cultural and linguistic formatting information described above is generally captured as a collection of data items and algorithms. This repository of cultural and linguistic formatting information is called a locale. Application developers use locales to internationalize/localize software. Locales are usually supplied by operating systems and can be found in both PC systems (e.g., OS/2, Unix, Windows, MacOS) and host based systems (e.g., OS/400 and System 390). It is becoming increasingly popular to find locales not only in operating systems but in programming languages (e.g., Java, Python, and Haskell) as well. In general there is little variation between the cultural formatting information found in the numerous locale implementations. [9],[108]

Locales are named according to the country and language they support. The name of a locale is based on two ISO (International Standards Organization) standards: ISO-639 (code for the representation of language names), and ISO-3166 (codes for the representation of names of countries). For the most part the name of a locale is formed by concatenating a code from ISO-639 with a code from ISO-3166. [95]

The setting of a locale is usually done by an individual user or a system administrator. When a system administrator configures a locale for a system, it is likely that they are selecting the default locale for an entire system. Depending on the system, a user may or may not have the ability to switch or modify a locale. [75]

2.5 Keyboard Input

Differences in writing systems also pose unique challenges. This is especially true for Chinese, Japanese, and Korean, because these languages have a large number of characters and thus require a special mechanism to input them from a keyboard. This special mechanism is known as an input method editor (IME). IMEs enable the

input of a large number of characters using a small set of physical keys on a keyboard. [75]

There are three basic types of IMEs: telegraph code, phonetic, and structural. A telegraph code IME allows the user to enter the decimal or hexadecimal value of a character rather than the actual character itself. This is a very simple to use IME, however, it requires the user to either memorize the decimal values of characters or carry around a book that lists all the values. [75]

A phonetic IME takes as input the phonetic representation of a character or sequence of characters rather than the characters themselves. The IME converts the phonetic representation into characters by using a dictionary. In some cases the phonetic representation may yield more than one result. When more than one result is obtained the user is given a list from which they make a selection for the appropriate conversion. The phonetic IME does not require a user to memorize decimal values, however it takes longer to determine the appropriate conversion due to dictionary lookup. [75]

In a structural IME users enter a character by selecting the character's building blocks. In many cases the same building blocks could generate more than one result. When multiple results are obtained the user is shown a list of candidates and is asked to pick the most appropriate character. Structural based IMEs do not require dictionary lookup, however they usually require the user to search through longer lists of characters. [75]

2.6 Fonts

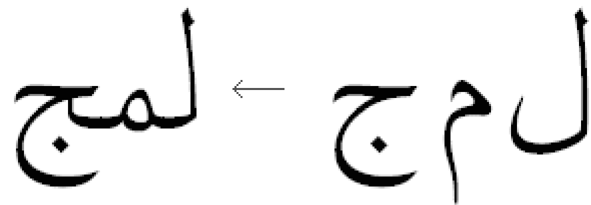
Users don't view or print characters directly, rather a user views or prints glyphs (a graphic representation of a character). Fonts are collections of glyphs with some element of design consistency. Many glyphs do not have a one-to-one relationship to characters. Sometimes glyphs represent combinations of characters. For

example, a user might type two characters, which might be displayed using a single glyph. See Figure 2-5. In other cases the choice of a glyph may be context dependent. For example, a character may take different forms depending on its position within a word: a separate glyph for a character at the beginning, middle, and end of a word. See Figure 2-6. [29],[69],[7]

Figure 2-5. Character to glyph mapping

f + i → ﬁ

Figure 2-6. Contextual glyphs



As scripts vary, so do the fonts that can display them. Assumptions that fonts of the same size and style will display different scripts approximately the same size is incorrect. This is especially true of fonts from Asia. Some scripts have constructs that go beyond simply placing one glyph after another in a row or column (e.g Japanese furigana, which are used for annotation). The furigana usually appear above ideographic characters and are used as a pronunciation guide. See Figure 2-7. Fully internationalized systems must take such issues into consideration. [29]

Figure 2-7. Japanese furigana characters

Recently there has been a strong push towards fonts with large glyph repertoire covering a wide variety of scripts. These fonts aim to simplify the construction of international software by providing both a uniform and consistent strategy for the presentation of character data. Such examples include: Apple's and Microsoft's TrueType font technology, Adobe's postscript fonts, and Apple's Advanced Type.

2.7 Character Coding Systems

One of the first steps in establishing an internationalized system is to modify programs so that they allow characters in a variety of languages and scripts. Software systems that support only a single language or script satisfies the needs of only a limited number of users. In computers scripts are captured through collections of discrete characters. Most people have an instinctive feeling for what a character is, rather than a precise definition. Naturally, many people would agree that the Latin alphabetic letters, Chinese ideographs, and digits are characters. The problems come when something looks like a single unit of a natural language but is actually comprised from multiple subpieces and when something is not really part of a natural language yet is actually represented with a character.

For many years the ASCII (American Standard Code for Information Interchange) encoding served as the definitive mechanism for storing characters. ASCII only allows for the encoding of 128 unique characters, due to the fact that ASCII uses only 7-bits for encoding. Languages that contain more than 128 unique characters require customized encoding schemes. This prevents efficient document exchange from occurring. Currently work by the Unicode Consortium is underway on the adoption of a character encoding standard, one that uses 16-bits greatly increasing the number of scripts that can be represented.

We select the area of character coding systems for further study because character coding systems serve as the foundation for multilingual information processing. As we will demonstrate the work in this area has not developed to a level of sophis-

tication for satisfying the needs of the multilingual information processing community. In this dissertation we recast character coding systems in a new light.

3 Character Coding Systems

In this chapter we explore the various approaches to encoding character data. We start the chapter by introducing terms relating to character coding systems followed by an examination of monolingual character coding systems, later turning our attention to multilingual coding systems. In this dissertation the term "monolingual encoding" refers to a character coding system that has strict limits on the number of scripts that can be represented, which are generally less than ten. We use the term "multilingual encoding" to refer to character coding systems that do not have strict limitations on the number of scripts that can be represented.

3.1 Terms

In this section, we define the terms related to character coding systems. The terminology used to describe character coding systems is often confusing, causing terms to be mistakenly used. The definitions of the terms are not formally standardized, therefore there can be some variation of terms across standards. In this dissertation we follow ISO (International Organization for Standardization) definitions where they exist. In cases where an ISO definition does not exist we use the definitions found in RFC 2130 and RFC 2277. RFC (Request For Comments) are published by the IETF (Internet Engineering Task Force). The IETF defines protocols that are used on the internet. Nevertheless, there are cases in which we choose to use alternative definitions over those found in RFC's. These alternative definitions are taken

from popular literature in the field of character encoding systems and are used when the popular definition is more widely accepted. [5],[64],[106]

The following terms and definitions are used in this dissertation:

- Character — “smallest component of written language with semantic value (including phonetic value).” [104]
- Control character — a character that affects the recording, processing, transmission or interpretation of data. [27]
- Graphic character — a character other than a control character, that has a visual representation. [27]
- Combining character — a member of an identified subset of a coded character set, intended for combination with the preceding or following character. [27]
- Glyph or glyph image— the actual concrete shape, representation of a character. [96],[106]
- Ligature — a single glyph that is constructed from one or more glyphs. [57]
- Octet — “an 8-bit byte.” [104]
- Character set — “a complete group of characters for one or more writing systems.” [104]
- Code point — “a numerical index (or position) in an encoding table (coded character set) used for encoding characters.” [96]
- Escape sequence — a sequence of bit combinations that is used for control purposes in code extension procedures. [27]
- Coded character set — “a mapping from a set of abstract characters to a set of integers (code points).” [104],[106]
- Code extension — a technique for encoding characters that are not included in a coded character set. [27]
- Character encoding scheme — “a mapping from a coded character set (or several) to a set of octets.” [104],[106]
- Code page — “a coded character set and a character encoding scheme which is part of a related series.” [104]
- Single byte character set (SBCS) — A character set whose characters are represented by one byte. [43]

- Double byte character set (DBCS) — A character set whose characters are represented by two bytes. [43]
- Multiple byte character set (MBCS) — A character set whose character are represented using a variable number of bytes. [43]
- Transport protocol — A data encoding solely for transmission purposes. [103]

3.2 Character Encoding Schemes

In general we divide character encoding schemes into two broad categories: monolingual and multilingual encodings. Additionally, character encoding schemes can be further divided into the following categories [64]:

- Fixed width encoding — in a fixed width encoding each character is represented using the same number of bytes.
- Modal encoding — in a modal encoding escape sequences are used to signal a switch between various character sets or modes.
- Non-modal or variable width encoding — in a non-modal encoding the code point values themselves are used to switch between character sets. Therefore, characters may be represented using a variable number of bytes. Characters in a non-modal encoding typically range from one to four bytes.

In the next section we study the character encoding schemes used predominantly in Western and Eastern Europe. We then turn our attention to the encoding systems used in Asia. In particular, we explore the encodings used to represent Japanese, Chinese, and Korean. Finally, we conclude the chapter with multilingual character encodings.

3.3 European Encodings

During the early 1960's the industry took the Latin alphabet, European numerals, punctuation marks, and various hardware control codes, and assigned them to a set of integers (7-bits) and called the resulting mapping ASCII (American Standard Code for Information Interchange.) ASCII is both a coded character set and an encoding. In ASCII each code point is represented by a fixed width 7-bit integer. It

is important to note that in ASCII characters are identified by their graphic shape and not by their meaning. Identifying characters by their shape simplifies character set construction. For example, in ASCII the code for an apostrophe is the same whether it is used as an accent mark, or as a single quotation mark. This simplification technique is certainly not a new idea. Typographers have been doing this for many years. For example, we no longer see Latin final character forms being used in English. [15],[26],[104],[87]

The ASCII encoding is divided into two broad sections: controls and graphic characters. ASCII controls are in the hex ranges 0x00-0x1F and 0x7F, while the ASCII graphic characters are in the hex range 0x20-0x7E. See Figure 3-1. ASCII's control codes are listed in Table 3-1. [15],[26],[104]

Figure 3-1. ASCII encoding

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	007F

Table 3-1. ASCII control codes

Value (hex)	Abbreviation	Name
0x00	NULL	null/idle
0x01	SOM	start of message

Table 3-1. ASCII control codes (Continued)

Value (hex)	Abbreviation	Name
0x02	EOA	end of address
0x03	EOM	end of message
0x04	EOT	end of transmission
0x05	WRU	"who are you..?"
0x06	RU	"are you..?"
0x07	BELL	audible signal
0x08	FE_0	format effector
0x09	HT/SK	horizontal tab/skip
0x0A	LF	line feed
0x0B	VTAB	vertical tabulation
0x0C	FF	form feed
0x0D	CR	carriage return
0x0E	SO	shift out
0x0F	SI	shift in
0x10	DC_0	data link escape
0x11	DC_1	device control
0x12	DC_2	device control
0x13	DC_3	device control
0x14	DC_4	device control stop
0x15	ERR	error
0x16	SYNC	synchronous idle
0x17	LEM	logical end of media
0x18	S_0	separator
0x19	S_1	separator
0x1A	S_2	separator
0x1B	S_3	separator
0x1C	S_4	separator
0x1D	S_5	separator
0x1E	S_6	separator

Table 3-1. ASCII control codes (Continued)

Value (hex)	Abbreviation	Name
0x1F	S_7	separator
0x7F	DEL	delete/idle

3.3.1 ISO 7-bit Character Encodings

Naturally, the number of languages that can be represented in ASCII is quite limited. Only English, Hawaiian, Indonesian, Swahili, Latin and some Native American languages can be represented. Even though ASCII is capable of representing English, ASCII at its core is uniquely American. For example, ASCII includes the dollar sign “$”, but no other currency symbol. So while British users can still use ASCII, they have no way of representing their currency symbol, the sterling “£”. [75],[104]

Seeing a need for representing other languages system vendors created variations on ASCII that included characters unique to specific languages. These variations are standardized in ISO-646. ISO-646 defines rules for encoding graphic characters in the hex range 0x20-0x7E. See Figure 3-1. Standards bodies then apply the rules to create language/script specific versions of ISO-646 sets. The ISO-646 standard defines one specific version that it calls the IRV (International Reference Version), which is the same as ASCII. [75],[46]

In ISO-646 a specific set of characters are guaranteed to be in every language/script specific version of ISO-646. These characters are referred to as the invariant characters. The invariant characters include most of the ASCII characters. Nevertheless, ISO also defines a set of characters that it calls variant characters.

Variant characters may be replaced by other characters that are needed for specific languages/scripts. See Table 3-2. [75],[46]

Table 3-2. ISO variant characters

Value (hex)	Character	Name
0x23	#	number sign
0x24	$	dollar sign
0x40	@	commercial at
0x5B	[	left square bracket
0x5C	\	backslash
0x5D	]	right square bracket
0x5E	^	circumflex
0x60	`	grave accent
0x7B	{	left curly bracket
0x7C	|	vertical bar
0x7D	}	right curly bracket
0x7E	~	tilde

There is no requirement that the variant characters be replaced in any of the language/script specific versions of ISO-646. For example, the French version of ISO-646 does not replace the circumflex "^" and the dollar sign "$". See Table 3-3. Unfortunately, French ISO-646 is still not capable of representing all of French in spite of these changes. In French all vowels can take circumflexes, however there is not enough room to represent them in ISO-646, hence the need for 8-bit encodings. [75], [46]

Table 3-3. French version of ISO-646

Value (hex)	Character	Name
0x23	£	sterling
0x24	$	dollar sign
0x40	à	a with grave

Table 3-3. French version of ISO-646 (Continued)

Value (hex)	Character	Name
0x5B	°	ring above
0x5C	ç	c with cedilla
0x5D	§	section sign
0x5E	^	circumflex
0x60	µ	micro sign
0x7B	é	e with acute
0x7C	ù	u with grave
0x7D	è	e with grave
0x7E	¨	diaeresis

3.3.2 ISO 8-bit Character Encodings

Around the late 1980s, the European Computer Manufacturers Association (ECMA) began creating and, with ISO, issuing standards for encoding European scripts based on an 8-bit code point. These standards have been adopted by most major hardware and software vendors. Such companies include IBM, Microsoft, and DEC [50]. The most popular and widely used of these encodings is ISO-8859-1, commonly known as Latin 1. [27],[104],[47]

The ISO-8859-1 encoding contains the characters necessary to encode the Western European and Scandinavian languages. See Figure 3-2 [44]. ISO-8859-1 is just one of many ISO-8859-x character encodings. Most of the ISO-8859-x encodings existed as national or regional standards before ISO adopted them. Each of the ISO-8859-x encodings have some common characteristics. Moreover, all of them function as both coded character sets and character encoding schemes. [26],[104],[47]

The ISO-8859-x encoding is divided into four sections. See Table 3-4. In each ISO-8859 encoding the first 128 positions are the same as ASCII. However, unlike ASCII the ISO-8859-x encodings rely on an 8-bit code point. Characters that

are required for specific scripts/languages appear in the 0xA0-0xFF range. This allows for the addition of 96 graphic characters over what ISO-646 IRV provides. Graphic characters that appear in multiple ISO-8859-x encodings all have the same code point value. For example, “é” appears in both ISO-8859-1 and ISO-8859-2 at the same code point value 0xE9.

Table 3-4. ISO-8859 layout

Code point range (hex)	Abbreviation	Name
0x00-0x1F	C0	ASCII controls
0x20-0x7E	G0	ISO-646 IRV
0x7F-0x9F	C1	control characters
0xA0-0xFF	G1	additional graphic characters

Figure 3-2. ISO-8859-1 encoding

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	007F
8-	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F
9-	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
A-	00A0	¡ 00A1	¢ 00A2	£ 00A3	¤ 00A4	¥ 00A5	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	ª 00AA	« 00AB	¬ 00AC	- 00AD	® 00AE	¯ 00AF
B-	° 00B0	± 00B1	² 00B2	³ 00B3	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	¹ 00B9	º 00BA	» 00BB	¼ 00BC	½ 00BD	¾ 00BE	¿ 00BF
C-	À 00C0	Á 00C1	Â 00C2	Ã 00C3	Ä 00C4	Å 00C5	Æ 00C6	Ç 00C7	È 00C8	É 00C9	Ê 00CA	Ë 00CB	Ì 00CC	Í 00CD	Î 00CE	Ï 00CF
D-	Ð 00D0	Ñ 00D1	Ò 00D2	Ó 00D3	Ô 00D4	Õ 00D5	Ö 00D6	× 00D7	Ø 00D8	Ù 00D9	Ú 00DA	Û 00DB	Ü 00DC	Ý 00DD	Þ 00DE	ß 00DF
E-	à 00E0	á 00E1	â 00E2	ã 00E3	ä 00E4	å 00E5	æ 00E6	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ì 00EC	í 00ED	î 00EE	ï 00EF
F-	ð 00F0	ñ 00F1	ò 00F2	ó 00F3	ô 00F4	õ 00F5	ö 00F6	÷ 00F7	ø 00F8	ù 00F9	ú 00FA	û 00FB	ü 00FC	ý 00FD	þ 00FE	ÿ 00FF

The C0 (0x00-0x1F) and C1 (0x7F-0x9F) ranges in ISO-8859-x are reserved for control codes. The C0 range contains the ASCII controls. The assignment of the control codes in the C1 range are not made in ISO-8859-x, but rather are part of ISO-6429 (control functions for 7-bit and 8-bit code sets). The motivation for the C1 controls comes from DEC's VT220 terminals.

Not all of the upper ranges G1 (0xA0-0xFF) of the ISO-8859-x standards are based on Latin characters. For example, ISO-8859-7 has Greek characters in the upper range. See Figure 3-3. The scripts/languages that the ISO-8859-x standards support is summarized in Table 3-5.[11],[75],[104]

Figure 3-3. ISO-8859-7 encoding

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	007F
8-	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F
9-	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
A-	00A0	‘ 2018	’ 2019	£ 00A3			¦ 00A6	§ 00A7	¨ 00A8	© 00A9		« 00AB	¬ 00AC	- 00AD		― 2015
B-	° 00B0	± 00B1	² 00B2	³ 00B3	΄ 00B4	΅ 0385	Ά 0386	· 0387	Έ 0388	Ή 0389	Ί 038A	» 00BB	Ό 038C	½ 00BD	Ύ 038E	Ώ 038F
C-	ΐ 0390	Α 0391	Β 0392	Γ 0393	Δ 0394	Ε 0395	Ζ 0396	Η 0397	Θ 0398	Ι 0399	Κ 039A	Λ 039B	Μ 039C	Ν 039D	Ξ 039E	Ο 039F
D-	Π 03A0	Ρ 03A1		Σ 03A3	Τ 03A4	Υ 03A5	Φ 03A6	Χ 03A7	Ψ 03A8	Ω 03A9	Ϊ 03AA	Ϋ 03AB	ά 03AC	έ 03AD	ή 03AE	ί 03AF
E-	ΰ 03B0	α 03B1	β 03B2	γ 03B3	δ 03B4	ε 03B5	ζ 03B6	η 03B7	ϑ 03B8	ι 03B9	κ 03BA	λ 03BB	μ 03BC	ν 03BD	ξ 03BE	ο 03BF
F-	π 03C0	ρ 03C1	ς 03C2	σ 03C3	τ 03C4	υ 03C5	φ 03C6	χ 03C7	ψ 03C8	ω 03C9	ϊ 03CA	ϋ 03CB	ό 03CC	ύ 03CD	ώ 03CE	

Table 3-5. ISO-8859 standards

Standard name	Informal name	Languages supported
ISO-8859-1	Latin 1	Afrikaans, Albanian, Basque, Catalan, Danish, Dutch, English, Faroese, Finnish, French, Gaelic, Galician, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, Swedish
ISO-8859-2	Latin 2	Albanian, Croatian, Czech, English, German, Hungarian, Polish, Romanian, Slovak, Slovenian
ISO-8859-3	Latin 3	Afrikaans, Catalan, Dutch, English, Esperanto, Galician, German, Italian, Maltese, Spanish, Turkish
ISO-8859-4	Latin 4	Danish, Estonian, English, Finnish, German. Lappish, Latvian, Lithuanian, Norwegian, Swedish
ISO-8859-5	Latin/Cyrillic	Azerbaijani, Belorussian, Bulgarian, English, Kazakh, Kirghiz, Macedonian, Moldavian, Mongolian, Russian, Serbian, Tadzhik, Turkmen, Ukrainian, Uzbek
ISO-8859-6	Latin/Arabic	Arabic, Azerbaijani, Dari, English, Persian, Farsi, Kurdish, Malay, Pashto, Sindhi, Urdu
ISO-8859-7	Latin/Greek	English, Greek
ISO-8859-8	Latin/Hebrew	English, Hebrew, Yiddish
ISO-8859-9	Latin 5	Turkish
ISO-8859-10	Latin 6	Greenlandic, Lappish
ISO-8859-11	Latin/Thai	English, Thai
ISO-8859-13	Latin 7	Baltic rim (region)
ISO-8859-14	Latin 8	Celtic
ISO-8859-15	Latin 9	removes some non character symbols from ISO-8859-1 adds "€" euro currency symbol.

3.3.3 Vendor Specific Character Encodings

Prior to the introduction of the ISO and European Computer Manufacturers Association character encoding standards, vendors created their own character encodings. For the most part these character encodings were created by hardware and operating system providers. In particular, IBM, Apple, Hewlett-Packard, and Microsoft. A large number of these vendor specific encodings are still in use. In this section we explore some of these encodings.

IBM has a number of code pages for its mainframe series of computers that are based upon EBCDIC (Extended Binary Coded Decimal Interchange Code). The IBM EBCDIC series of code pages use an 8-bit code point. In EBCDIC the number and type of printable characters are the same as ASCII, but the organization of EBCDIC differs greatly from ASCII. See Table 3-6 and Figure 3-4. [40],[57]

Table 3-6. EBCDIC layout

EBCDIC code point range (hex)	ASCII code point range (hex)	Characters
0x00-0x3F	0x00-0x1F	controls
0x40	0x20	space
0x41-0xF9	0x20-0x7E	graphic characters
0xFA-0xFE	N/A	undefined
0xFF	N/A	control

Figure 3-4. EBCDIC encoding

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	009C	0005	0086	0007	0097	008D	008E	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	009D	0085	0016	0087	0018	0019	0092	008F	001C	001D	001E	001F
2-	0080	0081	0082	0083	0084	000A	0017	001B	0088	0089	008A	008B	008C	0005	0006	0007
3-	0090	0091	0016	0093	0094	0095	0096	0004	0098	0099	009A	009B	0014	0015	009E	001A
4-	0020	00A0	â 00E2	ä 00E4	à 00E0	á 00E1	ã 00E3	å 00E5	ç 00E7	ñ 00F1	Ý 00DD	. 002E	< 003C	(0028	+ 002B	! 0021
5-	& 0026	é 00E9	ê 00EA	ë 00EB	è 00E8	í 00ED	î 00EE	ï 00EF	ì 00EC	ß 00DF	] 005D	$ 0024	* 002A	) 0029	; 003B	^ 005E
6-	- 002D	/ 002F	Â 00C2	Ä 00C4	À 00C0	Á 00C1	Ã 00C3	Å 00C5	Ç 00C7	Ñ 00D1	Š 0160	, 002C	% 0025	_ 005F	> 003E	? 003F
7-	ø 00F8	É 00C9	Ê 00CA	Ë 00CB	È 00C8	Í 00CD	Î 00CE	Ï 00CF	Ì 00CC	` 0060	: 003A	# 0023	@ 0040	' 0027	= 003D	" 0022
8-	Ø 00D8	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	« 00AB	» 00BB	ð 00F0	ý 00FD	þ 00FE	± 00B1
9-	° 00B0	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F	p 0070	q 0071	r 0072	ª 00AA	º 00BA	æ 00E6	ž 017E	Æ 00C6	€ 20AC
A-	µ 00B5	~ 007E	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	¡ 00A1	¿ 00BF	ð 00F0		Þ 00DE	® 00AE
B-	¢ 00A2	£ 00A3	¥ 00A5	· 00B7	© 00A9	§ 00A7	¶ 00B6	Œ 0152	œ 0153	Ÿ 0178	¬ 00AC	š 0161	ˉ 02C9		Ž 017D	× 00D7
C-	{ 007B	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	- 00AD	ô 00F4	ö 00F6	ò 00F2	ó 00F3	õ 00F5
D-	} 007D	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F	P 0050	Q 0051	R 0052	¹ 00B9	û 00FB	ü 00FC	ù 00F9	ú 00FA	ÿ 00FF
E-	\ 005C	÷ 00F7	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	² 00B2	Ô 00D4	Ö 00D6	Ò 00D2	Ó 00D3	Õ 00D5
F-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	³ 00B3	Û 00DB	Ü 00DC	Ù 00D9	Ú 00DA	009F

IBM also has a wide variety of IBM PC based code pages, sometimes referred to as DOS code pages. See Table 3-7 [35]. These code pages are also based upon an 8-bit code point. However, unlike EBCDIC, PC code pages are based upon ASCII. In particular, the hex range 0x00-0x7F is in direct correspondence with ASCII. For the most part the PC code pages are quite similar to the ISO-8859-x series. There are, however differences that are worth noting. For example, the IBM 850 PC code page

is quite similar to Latin 1, however IBM 850 assigns graphic characters to the 0x80-0xFF hex range, while ISO-8859-1 assigns control characters to the 0x80-0x9F hex range. Additionally, IBM 850 contains a small set of graphic box characters that are used for drawing primitive graphics. See Figure 3-5. [75]

Table 3-7. IBM PC code pages

Code page number	Language group
437	English, French, German, Italian, Dutch
850	Western Europe, Americas, Oceania
852	Eastern Europe using Roman letters
855	Eastern Europe using Cyrillic letters
857	Western Europe and Turkish
861	Icelandic
863	Canadian-French
865	Nordic
866	Russian
869	Greek

Figure 3-5. IBM 850

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		☺ 263A	☻ 263B	♥ 2665	♦ 2666	♣ 2663	♠ 2660	• 2022	◘ 25D8	○ 25CB	◙ 25D9	♂ 2642	♀ 2640	♪ 266A	♫ 266B	☼ 263C
1-	► 25BA	◄ 25C4	↕ 2195	‼ 203C	¶ 00B6	§ 00A7	▬ 25AC	↨ 21A8	↑ 2191	↓ 2193	→ 2192	← 2190	∟ 221F	↔ 2194	▲ 25B2	▼ 25BC
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	⌂ 2302
8-	Ç 00C7	ü 00FC	é 00E9	â 00E2	ä 00E4	à 00E0	å 00E5	ç 00E7	ê 00EA	ë 00EB	è 00E8	ï 00EF	î 00EE	ì 00EC	Ä 00C4	Å 00C5
9-	É 00C9	æ 00E6	Æ 00C6	ô 00F4	ö 00F6	ò 00F2	û 00FB	ù 00F9	ÿ 00FF	Ö 00D6	Ü 00DC	ø 00F8	£ 00A3	Ø 00D8	× 00D7	ƒ 0192
A-	á 00E1	í 00ED	ó 00F3	ú 00FA	ñ 00F1	Ñ 00D1	ª 00AA	º 00BA	¿ 00BF	® 00AE	¬ 00AC	½ 00BD	¼ 00BC	¡ 00A1	« 00AB	» 00BB
B-	░ 2591	▒ 2592	▓ 2593	│ 2502	┤ 2524	Á 00C1	Â 00C2	À 00C0	© 00A9	╣ 2563	║ 2551	╗ 2557	╝ 255D	¢ 00A2	¥ 00A5	┐ 2510
C-	└ 2514	┴ 2534	┬ 252C	├ 251C	─ 2500	┼ 253C	ã 00E3	Ã 00C3	╚ 255A	╔ 2554	╩ 2569	╦ 2566	╠ 2560	═ 2550	╬ 256C	¤ 00A4
D-	ð 00F0	Ð 00D0	Ê 00CA	Ë 00CB	È 00C8	€ 20AC	Í 00CD	Î 00CE	Ï 00CF	┘ 2518	┌ 250C	█ 2588	▄ 2584	¦ 00A6	Ì 00CC	▀ 2580
E-	Ó 00D3	ß 00DF	Ô 00D4	Ò 00D2	õ 00F5	Õ 00D5	µ 00B5	þ 00FE	Þ 00DE	Ú 00DA	Û 00DB	Ù 00D9	ý 00FD	Ý 00DD	¯ 00AF	´ 00B4
F-	- 00AD	± 00B1	‗ 2017	¾ 00BE	¶ 00B6	§ 00A7	÷ 00F7	¸ 00B8	° 00B0	¨ 00A8	· 00B7	¹ 00B9	³ 00B3	² 00B2	■ 25A0	00A0

In a similar fashion Microsoft has also created a series of code pages for use within Microsoft Windows. See Table 3-8 [35]. The Windows code pages are also

largely based upon ASCII and the ISO-8859-x series. The Windows code pages, however define extra graphic characters in the hex range 0x80-0x9F. See Figure 3-6.

Table 3-8. Windows code pages

Code page number	Languages supported
1252	Danish, Dutch, English, Finnish, French, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, Swedish
1250	English, Czech, Hungarian, Polish, Slovak
1251	Russian
1253	Greek
1254	Turkish

Figure 3-6. Windows 1252

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	007F
8-	€ 20AC	0081	‚ 201A	ƒ 0192	„ 201E	… 2026	† 2020	‡ 2021	ˆ 02C6	‰ 2030	Š 0160	‹ 2039	Œ 0152	008D	Ž 017D	008F
9-	0090	‘ 2018	’ 2019	“ 201C	” 201D	• 2022	– 2013	— 2014	˜ 02DC	™ 2122	š 0161	› 203A	œ 0153	009D	ž 017E	Ÿ 0178
A-	00A0	¡ 00A1	¢ 00A2	£ 00A3	¤ 00A4	¥ 00A5	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	ª 00AA	« 00AB	¬ 00AC	- 00AD	® 00AE	¯ 00AF
B-	° 00B0	± 00B1	² 00B2	³ 00B3	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	¹ 00B9	º 00BA	» 00BB	¼ 00BC	½ 00BD	¾ 00BE	¿ 00BF
C-	À 00C0	Á 00C1	Â 00C2	Ã 00C3	Ä 00C4	Å 00C5	Æ 00C6	Ç 00C7	È 00C8	É 00C9	Ê 00CA	Ë 00CB	Ì 00CC	Í 00CD	Î 00CE	Ï 00CF
D-	Ð 00D0	Ñ 00D1	Ò 00D2	Ó 00D3	Ô 00D4	Õ 00D5	Ö 00D6	× 00D7	Ø 00D8	Ù 00D9	Ú 00DA	Û 00DB	Ü 00DC	Ý 00DD	Þ 00DE	ß 00DF
E-	à 00E0	á 00E1	â 00E2	ã 00E3	ä 00E4	å 00E5	æ 00E6	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ì 00EC	í 00ED	î 00EE	ï 00EF
F-	ð 00F0	ñ 00F1	ò 00F2	ó 00F3	ô 00F4	õ 00F5	ö 00F6	÷ 00F7	ø 00F8	ù 00F9	ú 00FA	û 00FB	ü 00FC	ý 00FD	þ 00FE	ÿ 00FF

Hewlett-Packard has also codified a set of 8-bit code pages that are very much akin to the ISO-8859-x series. In particular, Hewlett-Packard’s ROMAN8. In ROMAN8 the hex range 0x00-0x7F is in direct correspondence with ASCII. ROMAN8 provides some additional characters that are not in ISO-8859-1. Nevertheless, ROMAN8 does not have some characters that are in ISO-8859-1. For example, the “©” copyright sign is present in ISO-8859-1, but not in ROMAN8. [63],[75]

3.4 Japanese Encodings

Languages like Japanese, Chinese, and Korean have extensive writing systems that are based on both phonetic and ideographic characters. These systems require more than 8-bits per character, because they contain literally thousands of characters. In this section we describe the encoding methods used in Japan. We choose to use Japanese for two reasons: First, Japan was the first country to encode a large character set. Second, the Chinese and Korean encodings are largely based upon the methods used in the Japanese encodings. [57]

The Japanese writing system is comprised from four different script systems: Chinese ideographic characters (symbols that represent ideas or things, known as *Kanji* in Japan), *Katakana* (syllabary used for writing borrowed words from other languages), *Hiragana (*syllabary used for writing grammatical words and inflectional endings), and *Romaji* (Latin letters used for non Japanese words). The Kanji script used in Japan includes about 7,000 characters. The Katakana and Hiragana scripts both represent the same set of 108 syllabic characters, and are collectively known as *Kana*. A sampling of Kanji, Katakana, and Hiragana characters are shown in Figure 3-7, Figure 3-8, and Figure 3-9. [57],[73],[96],[109]

Figure 3-7. Japanese Kanji characters

丁 4E01	丑 4E11	両 4E21	丱 4E31	乁 4E41	乑 4E51	乡 4E61	乱 4E71
丂 4E02	丒 4E12	丢 4E22	串 4E32	乂 4E42	乒 4E52	乢 4E62	乲 4E72
七 4E03	专 4E13	丣 4E23	丳 4E33	乃 4E43	乓 4E53	乣 4E63	乳 4E73

Figure 3-8. Japanese Katakana characters

ア A	イ I	ウ U	エ E	オ O
カ KA	キ KI	ク KU	ケ KE	コ KO
サ SA	シ SHI	ス SU	セ SE	ソ SO
タ TA	チ CHI	ツ TSU	テ TE	ト TO
ナ NA	ニ NI	ヌ NU	ネ NE	ノ NO
ハ HA	ヒ HI	フ FU	ヘ HE	ホ HO
マ MA	ミ MI	ム MU	メ ME	モ MO
ヤ YA		ユ YU		ヨ YO
ラ RA	リ RI	ル RU	レ RE	ロ RO
ワ WA				ヲ WO
ン N				

Figure 3-9. Japanese Hiragana characters

あ A	い I	う U	え E	お O
か KA	き KI	く KU	け KE	こ KO
さ SA	し SHI	す SU	せ SE	そ SO
た TA	ち CHI	つ TSU	て TE	と TO
な NA	に NI	ぬ NU	ね NE	の NO
は HA	ひ HI	ふ FU	へ HE	ほ HO
ま MA	み MI	む MU	め ME	も MO
や YA		ゆ YU		よ YO
ら RA	り RI	る RU	れ RE	ろ RO
わ WA				を WO
ん N				

When discussing systems for encoding Japanese, we must be careful to make the distinction between Japanese coded character sets and Japanese character encodings. In Japanese the coded character set is referred to as the Ward-Point or Kuten character classification system. This system is part of the JIS X (Japan Industrial

Standard) published by the Japanese Standards Association. Using the Ward-Point system requires knowledge of spoken and written Kanji. The system uses a two code system for character identification. The two codes are commonly known as row and cell and are represented using two bytes. [86]

Using a single byte 128 positions can be referenced. Nevertheless, the Japanese only use the 94 printable ASCII characters. Therefore, in the Ward-Point system, Kanji characters are arranged in groups of 94 characters. Each character is assigned two values, a row number and a cell number. These values identify the character's position within JIS. The row and cell numbers range from 1 to 94. Therefore, JIS can be thought of as 94x94 character matrix. This allows 8,836 characters to be represented. This coded character set scheme was first formalized in 1978 and is known as JIS X 0208. [86]

In Japanese characters may be read in one of two ways, *On* and *Kun*. On is a reading that is based on a Chinese pronunciation, while *Kun* is a reading based on a Japanese pronunciation. Radicals in Japanese represent core components of Kanji characters. These radicals are indicative of a group of Kanji characters. [86]

In the Ward-Point system characters are grouped according to their reading and their radical makeup. Additionally, JIS specifies three levels of character groupings. Level 0 contains non-Kanji characters, while levels 1 and 2 are used exclusively for Kanji characters. Level 1 contains the 2,965 most frequently occurring characters, while level 2 contains an additional 3,388 characters. [86],[105]

The Japanese Standards Association has continually revised JIS adding, removing and rearranging characters. Besides the Kanji characters, JIS also encodes the Latin, Greek, and Cyrillic alphabets, because of business needs. The various JIS

standards along with the characters that they each encode is summarized in Table 3-9. [58],[86]

Table 3-9. JIS character standards

Standard name	Year adopted	Number of characters	Characters
JIS X0201-1976	1976	128	Latin, Katakana
JIS X0208-1978	1978	6,879	Kanji, Kana, Latin, Greek, Cyrillic
JIS X0208-1983	1983	6,974	Kanji, Kana, Latin, Greek, Cyrillic
JIS X0208-1990	1990	6,976	Kanji, Kana, Latin, Greek, Cyrillic
JIS X0212-1990	1990	6,067	Kanji, Greek with diacritics, Eastern Europe, Latin
JIS X0213-2000	2000	4,344	Kanji, Kana, Latin, Greek, Cyrillic

As we mentioned earlier the Japanese coded character sets are independent from the method used to encode them. In the next section we examine the four methods (Personal Computer, Extended Unix Code, Host, and Internet Exchange) used for encoding Japanese. The main difference between these encoding methods is in the way in which they switch between single byte character sets (SBCS) and double byte character sets (DBCS).

3.4.1 Personal Computer Encoding Method

The Personal Computer (PC) encoding method is a non-modal system. In a non-modal encoding the code point value of a character is used to switch between SBCS (ASCII characters) and DBCS (JIS Kanji). This encoding system is generally referred to as shift-JIS (SJIS), because the Kanji characters shift around the SBCS characters. In the SJIS encoding system DBCS mode is initiated when the code point value of the first character of a two byte sequence falls within a predefined range above hex 0x7F. Generally the range is hex 0x81-0xFF or 0xA1-0xFE. If a character falls in the two byte range, then the character is treated as the first half of a two byte sequence, otherwise the character is treated as a single byte sequence. See Figure 3-10. The code page illustrated in Figure 3-11 represents JIS X0212 in this code page

there are two double byte ranges hex 0x81-0x9F and hex 0xE0-0xFC. Additionally, in JIS X0212 the Katakana characters are encoded as single byte sequences. See Figure 3-11 hex range 0xA1-0xDF. [43],[86]

Figure 3-10. Mixed DBCS and SBCS characters

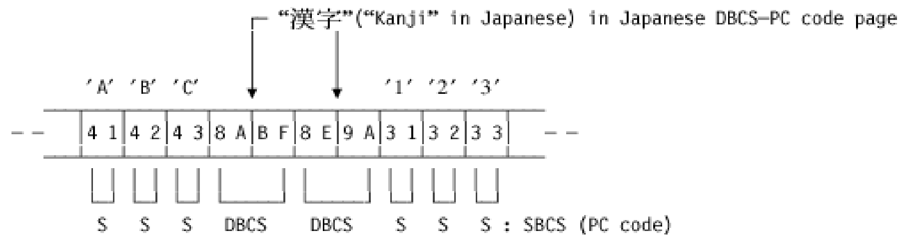


Figure 3-11. JIS X0212 PC encoding

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	¥ 00A5	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	‾ 203E	0080
8-		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
9-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
A-		｡ FF61	｢ FF62	｣ FF63	､ FF64	･ FF65	ｦ FF66	ｧ FF67	ｨ FF68	ｩ FF69	ｪ FF6A	ｫ FF6B	ｬ FF6C	ｭ FF6D	ｮ FF6E	ｯ FF6F
B-	ｰ FF70	ｱ FF71	ｲ FF72	ｳ FF73	ｴ FF74	ｵ FF75	ｶ FF76	ｷ FF77	ｸ FF78	ｹ FF79	ｺ FF7A	ｻ FF7B	ｼ FF7C	ｽ FF7D	ｾ FF7E	ｿ FF7F
C-	ﾀ FF80	ﾁ FF81	ﾂ FF82	ﾃ FF83	ﾄ FF84	ﾅ FF85	ﾆ FF86	ﾇ FF87	ﾈ FF88	ﾉ FF89	ﾊ FF8A	ﾋ FF8B	ﾌ FF8C	ﾍ FF8D	ﾎ FF8E	ﾏ FF8F
D-	ﾐ FF90	ﾑ FF91	ﾒ FF92	ﾓ FF93	ﾔ FF94	ﾕ FF95	ﾖ FF96	ﾗ FF97	ﾘ FF98	ﾙ FF99	ﾚ FF9A	ﾛ FF9B	ﾜ FF9C	ﾝ FF9D	ﾞ FF9E	ﾟ FF9F
E-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
F-	X	X	X	X	X	X	X	X	X	X	X	X	X			

3.4.2 Extended Unix Code Encoding Method

The extended Unix Code (EUC) system is used predominately in Unix environments. EUC consists of four graphic character code sets: the primary graphic code set (G0) and three supplementary code sets (G1, G2, and G3). Code set G0 is always a single byte per character code set, and is usually ASCII. Code set G1, usually Kanji

consists of two byte per character sequences that have their most significant bits set. Characters in code set G2 are required to have their byte sequence start with a special prefix, hex 0x8E. This special prefix is known as single-shift-two (SS2). Characters in G2 are often katakana characters. Characters in code set G3 must also have their byte sequence start with a special prefix, however in the case of G3 the prefix is hex 0x8F, known as single-shift-three (SS3). The G3 code set is reserved for user defined or external characters.

In EUC Kanji mode is initiated when the value of the first character of a two byte sequence is between hex 0xA1-0xFE. This character is then treated as the first half of a two byte sequence. The second byte from this sequence must also be in the same range. The ASCII mode is initiated when the first character is less than hex 0x7F. Katakana mode is invoked when the first character is SS2. This character is subsequently treated as the first half of a two byte character sequence. Additionally, the second byte must be in the range hex 0xA1-0xDF. The user defined character mode is initiated when the first character is SS3. This character is subsequently treated as the first byte of a three byte character sequence. The second and third bytes must come form the range hex 0xA1-0xFE. The example in Figure 3-12 demonstrates EUC with one, and two byte sequences. Line 1 on Figure 3-12 are hex byte sequences, separated by commas that correspond to the graphic characters on Figure 3-12. Additionally, the underlined byte sequences indicate double byte sequences

Figure 3-12. EUC encoding

ABC漢字 123

41,42,43,B4C1,BBFA,20,31,32,33 **(1)**

3.4.3 Host Encoding Method

On host environments, such as the IBM 390 and AS/400 both SBCS and DBCS are used. In the host method special control codes are used to switch between

SBCS mode and DBCS mode. To switch into DBCS mode the shift-in (0x0E) control is used, while the shift-out (0x0F) control is used to switch back to SBCS mode. Unlike the PC encoding method the values of the characters themselves are not used to determine the mode, therefore the host method is a modal encoding system. On host systems SBCS is based on EBCDIC, while DBCS is based on JIS. [43]

3.4.4 Internet Exchange Method

The internet exchange method, commonly known as ISO-2022 is a 7-bit/8-bit encoding method that enables character data to be passed through older systems. In some legacy systems the high order bit of an 8-bit byte gets stripped off, causing corruption of 8-bit character data. Therefore, all bytes in ISO-2022 must be in the hex range 0x21-0x7E (printable ASCII). Generally, ISO-2022 is never used as an internal character encoding. Nevertheless, Emacs processes character data in the ISO-2022 encoding [88]. The use of the term ISO-2022, however is somewhat misleading, because ISO-2022 does not really specify character encodings, but rather an architecture for intermixing coded character sets. The actual individual encodings are specified in RFCs. [104]

In ISO-2022 escape sequences and shift states are used to switch between coded character sets. Therefore, ISO-2022 is a modal encoding system. ISO-2022 reserves the hex range 0x00-0x1F for 32 control codes and refers to this range as the C0 block. This is the same as the ISO-8859 standard. Additionally, another set of 32 controls are also reserved, designated as C1. These controls may be represented with escape sequences. The hex range 0x20-0x7F is reserved for up to four sets of graphic characters, designated G0-G3 (in some graphic sets, each character may require multiple bytes). Most graphic sets only use the hex range 0x21-0x7E, in which case 0x20 (space), and 0x7F (delete) are reserved. Typically, the C0 and C1 blocks are taken from ISO-6429. See Table 3-1. The G0 block is generally taken from ISO-646 (International Reference Version). See Figure 3-1. In ISO-2022 an escape sequence starts

with an ESC control character (0x1B). The bytes following the ESC are a set of fixed values that are defined by the ISO-2022 standard. [24]

In many cases a single stream of characters can be encoded in more than one way in ISO-2022. For example, the mixed ASCII Greek character stream on line 1 on Figure 3-13 can be represented by switching graphic character sets or by escaping individual characters. Line 2 is the corresponding 7-bit byte sequence for the characters on line 1. The double underlined characters on line 2 indicate escape sequences. By default ISO-2022 sets the G0 graphic character set to ASCII. Escape sequences are used to switch character sets. The first escape sequence “1B,2C,46” on line 2 switches the G0 graphic character set to 7-bit Greek, while the second escape sequence “1B,28,42” switches G0 back to ASCII. Line 3 is the corresponding 8-bit byte sequence for the characters on line 1. The double underlined characters on line 3 indicate an escape sequence, while the single underlined characters “8E” indicate a single-shift-two (SS2). The escape sequence “1B,2E,46” on line 3 assigns the 8-bit Greek character set to the G2 graphic character set, while the SS2 character signals a temporary switch into the G2 character set.

Figure 3-13. ISO-2022 encoding

Greek Ελληνικά **(1)**

47,72,65,65,6B,20,1B,2C,46,45,6B,6B,67,6D,69,6A,6C,1B,28,42 **(2)**

47,72,65,65,6B,20,1B,2E,46,8E,C5,8E,EB,8E,EB,8E,E7,8E,ED,8E,E9,8E,EA,8E,DC **(3)**

In the case of ISO-2022-JP (Japanese encoding of ISO-2022), a Kanji-in escape sequence directs the bytes that follow to be treated as two bytes per character. The first byte of the two byte sequence determines the character grouping, while the second byte indicates the character within the grouping. A JIS out or Kanji out escape sequence directs the bytes that follow to be treated as single byte characters. The escape sequences for ISO-2022-JP are specified in Table 3-10. The example, on

Figure 3-14 illustrates how the characters from Figure 3-12 would be represented in ISO-2022-JP. Line 1 are hex byte sequences that correspond to the graphic characters on Figure 3-14. Double underlined bytes represent escape sequences, while underlined bytes indicate double byte character sequences. [86],[104]

Table 3-10. ISO-2022-JP escape sequences

Escape sequence (hex)	Escape sequence (graphic)	Coded character set
0x1B,0x28,0x42	ESC (B	ASCII
0x1B,0x28,0x4A	ESC (J	JIS X0201 (Roman)
0x1B,0x24,0x40	ESC $ @	JIS X0208-1978
0x1B,0x24,0x42	ESC $ B	JIS X0208-1983

Figure 3-14. ISO-2022-JP encoding

ABC漢字 123

41,42,43,1B,24,42,3441,3B7A,1B,28,42,20,31,32.33 **(1)**

3.4.5 Vendor Specific Encodings

IBM, Microsoft, Apple, and DEC have provided a wide variety of Japanese character encodings for use within their respective platforms and operating systems. Unfortunately, in many cases these encodings are incompatible across vendors. Naturally, as national/international standards emerged some migration path from legacy encodings to standardized encodings became necessary. To satisfy this need vendors have created/modified encodings. In almost all cases these encodings are variations or super sets of existing encodings, although differences do exist. Some of the more frequently occurring encodings are listed on Table 3-11. [52],[42],[43]

Table 3-11. Vendor encodings

Vendor	Code page	Based on
IBM	952	DBCS EUC JISX0208-1997
IBM	953	DBCS EUC JISX0212-1990

Table 3-11. Vendor encodings (Continued)

Vendor	Code page	Based on
IBM	932/942	MBCS PC JIS X0208-1978
IBM	943	MBCS PC JIS X0208-1990
Microsoft	MS 932	MBCS PC JIS X0208-1990
HP	HP EUC	DBCS EUC JIS X0212-1990
DEC	DEC Kanji	ISO-2022-JP JIS X0208-1983

3.5 Chinese Encodings

The traditional Chinese writing system is not a phonemic system. In Phonemic systems sound is represented as units that are in a one-to-one correspondence with symbols. For example, a Latin based language, can be written with only 26 unique letters. On the other hand, Chinese requires thousands of unique symbols (ideographic characters, known as Hanzi in China) to express itself. Moreover, it is difficult to determine just how many characters exist today. One of the classic Chinese dictionaries lists about 50,000 Hanzi characters, of which only 2,000-3,000 are in general use. [93]

In general Chinese Hanzi characters are difficult to write and print due to the large number of strokes in each character. Hanzi characters vary from one to over 30 strokes. Typically, each character requires seven to 17 strokes. The practical disadvantages of such a system when compared to an alphabet are obvious. [93]

Over time, however the Chinese script has taken on certain phonetic properties. In some cases, identical sounding but semantically remote characters would loan their shapes to indicate the sound of a character. Additionally, the Chinese script and language have been in a continuos state of flux. In particular, since the revolution of 1949 the Chinese government (Peoples Republic of China) has actively pursued simplification of the Chinese script. [93]

In 1954 a committee was formed to reform the language. This committee simplified nearly 2,200 Hanzi characters. In some cases the radicals (base shape of a

Chinese Hanzi character) changed, while in others the number of strokes changed. This simplification, however caused havoc in dictionaries, because Chinese dictionaries are organized by radical and stroke. Some scholars believe that the simplification process has only made the Chinese script more difficult to understand. Specifically, the reduction in the number of strokes makes several characters look alike. While the government of the Peoples Republic of China has continued to promote this simplification process, it has not been universally accepted however, particularly in Taiwan and Hong Kong [93]

The Chinese language can also be represented through transliteration. In this context we refer to transliteration of Chinese into its phonetic equivalent in Latin letters. In general there are two Latin transliteration systems for Chinese, Wade-Giles and Pinyin. The Wade-Giles system was invented by two british scholars during the 19th century. The Wade-Giles system is only used in Taiwan for representing place names, street names, and people's names. In mainland China only Pinyin is used. [93],[77]

The Pinyin system was created during the Chinese Hanzi simplification process. The Pinyin system can be used with or without diacritics. Most systems opt for using Pinyin without diacritics, because diacritics require special fonts. [93]

In Taiwan the Bopomofo system is used for transliteration. The Bopomofo system gets its name from the first four Taiwanese phonetic characters. The Bopomofo characters represent consonants and vowels. Moreover, there is a one-to-one correspondence between Pinyin and Bopomofo. [93],[77]

3.5.1 Peoples Republic of China

As we indicated above Japan was the first country to construct a large coded character set and encoding. Similarly, China has done the same in their *Guojia Biaozbun* (GB) standards; *Guojia Biaozbun* means National Standard. The Chinese use the GB 2312-80 standard to manage Hanzi (Simplified Chinese characters),

Bopomofo, Pinyin, Japanese Katakana, Japanese Hiragana, Latin, Greek, and Cyrillic in groups of 94x94 character matrices. This 94x94 matrix is the same design employed in the Japanese standards. The overall structure is similar to JIS, but the Chinese characters (Hanzi) are placed in different positions. The non-Hanzi characters are in the same locations as JIS. [104],[48],[105]

Just like JIS, the Hanzi characters are organized into two levels based upon their frequency of use. Two additional groups for even less frequently used Hanzi characters have been developed, for a total of three groups of Hanzi characters. Additionally, GB 2312-80 may be encoded using the PC, EUC, ISO-2022, and Host encoding methods. [104],[48],[105]

After the construction of the GB 2312-80 standard, the Peoples Republic of China expressed interest in supporting the efforts of both the Unicode Consortium and ISO through publishing a Chinese national standard that was code and character compatible with the evolving ISO-10646/Unicode standard, in particular version 2.1 of the Unicode standard. We delay a detailed discussion of ISO-10646/Unicode until later. For purposes of discussion we can think of Unicode as a super set of all coded character sets. [67]

This new Chinese standard was named GB 13000.1-93, which is commonly known as GB 13000. Whenever ISO/Unicode would change their standard, the Chinese would also update their standard. By adopting this strategy, GB 13000 was able to include Traditional Chinese Hanzi characters (ideographic characters used in Taiwan and Hong Kong), because these characters appeared in Unicode. Unfortunately, GB 13000's character encoding was not compatible with GB 2312-80. In order to remain compatible with the GB 2312-80 encoding standard a new coded character set was created that contained all the characters from both GB 13000 and GB 2312-80, yet used an encoding that was compatible with GB 2312-80. This new character set is known as *Guojia Biaozbun Kuozban* (GBK) and also uses groups of

94x94 character matrices. Thus, code and character compatibility between GB 2312-80 and GBK was ensured while at the same time, remaining synchronized with Unicode's character set. [67],[94]

Prior to the release of the Unicode 3.0 standard, GBK was regarded as the de facto coded character set and encoding for Mainland China. As the Unicode standard progressed GBK became full. Finally, when it became time to adopt the new characters in Unicode 3.0 GBK would have to expand to a three byte per character encoding. Thus, a new coded character set and character encoding was born. This new encoding is known as GB 18030. GB 18030 is a multi byte encoding (one to four bytes). The one and two byte portions, however are compatible with GBK. GB 18030 thus creates a one-to-one relationship between parts of GB 18030 and Unicode's encoding space. We summarize the various GB standards in Table 3-12. [59],[67],[84]

Table 3-12. GB standards

Standard name	Year adopted	Number of characters	Characters
GB 2312-80	1981	7,445	Simplified Hanzi,Traditional Hanzi (some), Pinyin, Bopomofo, Hiragana, Katakana, Latin, Greek, Cyrillic
GBK	1993	21,886	Simplified Hanzi,Traditional Hanzi (some), Pinyin, Bopomofo, Hiragana, Katakana, Latin, Greek, Cyrillic
GB 18030-2000	2000	28,468	Simplified Hanzi,Traditional Hanzi (some), Pinyin, Bopomofo, Hiragana, Katakana, Latin, Greek, Cyrillic

3.5.2 Republic of China

Taiwan has developed Chinese National Standard (CNS) 11643, which contains over 48,000 Traditional Chinese characters plus characters from the various other scripts. CNS 11643 is also organized into groups of 94x94 character matrices. Nevertheless, CNS 11643 is not the predominant coded character set within Taiwan, rather BigFive is used. BigFive refers to the five companies that created it. BigFive

is grouped into 94x157 character matrices. BigFive encodes over 13,000 Traditional Chinese characters plus Latin, Greek, Bopomofo, and other symbols. Fortunately there are few differences between BigFive and CNS 11643. We summarize the Taiwanese standards in Table 3-13. [104]

Table 3-13. Taiwanese standards

Standard name	Year adopted	Number of characters	Characters
CNS 11643	1992	48,027	Traditional Chinese Hanzi, Bopomofo, Latin. Greek
BigFive	1984	13,494	Traditional Chinese Hanzi, Bopomofo, Latin. Greek

3.5.3 Hong Kong Special Administrative Region

Historically, computers lacked support for the special characters commonly used in Hong Kong and in the areas where Cantonese is spoken. Some of the missing Hanzi characters were of foreign origin, particularly deriving from Japanese. The use of these characters reflects Hong Kong's role in the economics of Asia. Neither, the BigFive, GB 2312-80, or GBK adequately supported the needs of Hong Kong or general Cantonese users. [67],[68]

Software vendors created various solutions to providing the missing characters. Unfortunately, their efforts were uncoordinated resulting in solutions that were incompatible with each other. Concurrent to this activity the special administrative government of Hong Kong started developing the "Hong Kong Government Chinese Character Set". This informal specification was initially used internally as a governmental standard. Soon after, it became a required feature for general computing systems within Hong Kong. [67],[68]

In 1999 these special characters were officially published in the "Hong Kong Supplementary Character Set" (SCS). The SCS contains 4,072 characters, the majority of which are Hanzi. Additionally, the SCS was explicitly designed to fully

preserve the code point organization of BigFive, thus easing the problem of encoding. [67],[68]

3.6 Korean Encodings

Just like Japanese and Chinese, Korean also uses a set of ideographic characters in its writing system. These ideographic characters are known as *Hanja* in Korean. Additionally, Korean also uses a set of phonetic characters, referred to as *Hangul*. The Hangul script was created by royal decree in 1443 by a group of scholars. Each Hangul character is a grouping of two to five Hangul letters, known as Jamo (phonemes). Each Hangul block forms a square cluster representing a syllable in the Korean language. Jamo can be simple or double consonants and vowels. The modern Hangul alphabet contains 24 basic Jamo elements (14 consonants and 10 vowels). Extended letters can be derived by doubling the basic letters. [6],[109]

3.6.1 South Korea

As in the other Asian encodings Korea's standards (KS) follow a layout similar to JIS and GB. However, Cyrillic and Greek characters are not in the same positions as JIS and GB. In the KSX-1001 (formerly KSC-5601) characters are organized into 94x94 matrices. KSX-1001 encodes Jamos (Korean letters), Hangul, Hanja, Katakana, Hiragana, Latin, Greek, Cyrillic, as well as other symbols. Additionally, Korea also encodes their own version of the ISO-646 standard, replacing hex 0x5C (backslash) with the Won sign (Korean currency symbol). KSX-1001 can be encoded using EUC and ISO-2022. [104]

3.6.2 North Korea

The North Korean government has also created a coded character set for Korean, known as KPS 9566-97. It is constructed in a similar fashion to South Korea's KSX-1001. KPS 9566-97 encodes nearly 8,300 characters including: Jamos,

Hangul, Hanja, Latin, Cyrillic, Greek, Hiragana, and Katakana. It can be encoded by using EUC and ISO-2022. [22]

3.7 Vietnamese Encodings

Vietnamese was first written using the Chinese ideographic characters. This system was in use by scholars until a few decades ago. In Vietnamese two Chinese characters were usually combined, one character indicated the meaning, while the second assisted with pronunciation. This system, *chu nom,* never gained widespread adoption and was only used in literature. [92]

Around the 17th century Catholic missionaries arrived in Vietnam and began to translate prayer books. In doing their translations they developed a new Romanized script. This script is known as *quoc ngu.* Initially this new script was not met with mass appeal. Nevertheless, when Vietnam became under French control (1864-1945) quoc ngu was officially adopted. Thus, in modern Vietnam quoc ngu is used universally and forms the basis for all Vietnamese computing. [92]

It would appear that modern Vietnamese could easily be incorporated into one of the Latin based encodings, as Vietnamese is based upon a French model of Latin characters. Like French an 8-bit encoding scheme should be sufficient for encoding Vietnamese. Nevertheless, in Vietnamese there are many frequently occurring accented letters. In addition to the alphabetic characters in the IRV (ASCII 0x00-0x7F), Vietnamese requires an additional 134 combinations of a letter and diacritical symbols. [92]

Obviously all such combinations can fit within the confines of an 8-bit encoding space. However, it is highly desirable to maintain compatibility with the IRV range. Requiring such compatibility does not leave enough room in the upper range (0x80-0xFF) to encode all the necessary diacritic combinations. Some people within the Vietnamese data processing community have argued that certain rarely used

precomposed Vietnamese characters could be dropped altogether or could be mapped into the C0 control space (0x00-0x1F). [92]

Until the introduction of Windows 95, no clear encoding standard had emerged. Microsoft and the Vietnam Committee on Information Technology created a new code page, known as Microsoft 1258 (CP 1258). CP 1258 can be made compatible with Latin 1 by dropping some precomposed characters. There are other competing standards emerging, however. Most notably is the VISCII (Vietnamese Standard Code for Information Interchange) standard, described in RFC 1456. The VISCII standard does preserve all the precomposed characters, and is becoming quite popular. [92]

In addition to the Microsoft and VISCII encoding schemes, there is a convention for exchanging Vietnamese across 7-bit systems. This 7-bit convention is known as VIQR (Vietnamese Quoted-Readable), and is described in RFC 1456. VIQR is not really encoding scheme, but is rather a method for typing, reading, and exchanging Vietnamese data using ASCII. In VIQR precomposed characters are represented by the vowel followed by ASCII characters whose appearances resemble those of the corresponding Vietnamese diacritical marks. [92]

3.8 Multilingual Encodings

So far in our discussion of encoding schemes we have concentrated our efforts on monolingual encodings. In this section we turn our attention to multilingual encodings. We start this section with some motivation for the construction of multilingual encodings, later turning our attention to the various strategies for capturing multilingual data.

3.8.1 Why Are Multilingual Encodings Necessary?

Over the last twenty years the software industry has experienced incredible growth. Initially, the demand for software was limited to just the United States,

however as cheap computing proliferated this demand has spread the world over. For the most part the development of software has been restricted to the United States. Historically, most software development labs would produce an English language version of a product, subsequently followed by multiple national language versions (NLV). The construction of these NLVs was generally not done in the United States, rather it was done by the overseas branch of the development lab or was contracted out to an independent software vendor (ISV). [29],[20],[85]

Once the source code was delivered to the overseas lab or ISV the source code would be modified to support the local encoding schemes for the language/country. Simultaneous to this effort, a lab in the United States would start development on the next version of the product. Therefore, the various NLVs always lagged behind the English version of the product. [29],[61],[85] This situation caused several problems:

- Overseas marketing organizations faced a difficult time selling older versions of products when newer versions were available in the United States.
- It became difficult to provide timely maintenance to a product because a fix would need to be generated across several source trees.
- In many cases common fixes could not be used across source lines because each source tree supported a different coded character set and encoding.
- Attempting to later merge support for all encoding schemes across all source trees dramatically increased both the size and the complexity of the product.

Differences in encoding approaches and text processing make merging source trees extremely difficult. In some situations merging source trees requires text processing algorithms to be rewritten. For example, random character access functions may need to be rewritten when a stateful encoding is merged with a stateless encoding. In a stateful encoding the meaning of a character is dependent on neighboring characters.

In some cases even memory management routines require modification. In fixed width encoding schemes developers often assume that a byte and a character are of the same size, or even worse the number of code units in a string represents the

number of characters in the string. These assumptions causes problems for merging source code based on a fixed width encoding scheme with source code based on a variable width encoding scheme.

Merging problems may also be caused by differences in encoding philosophies. In some cases entire text processing functions may need to be either removed or redesigned when source trees are merged. For example, a text searching function based on abstract characters would have to be redesigned when used with an encoding based on glyphs.

These problems caused the software industry to reach two important conclusions: First, that there was an imperative need to develop a single worldwide coded character set and encoding that would be required for all software. Second, that all NLVs of a product must be based on a single common source tree.

3.8.2 Unicode and ISO-10646

In discussing the creation of Unicode, we can not avoid discussing ISO-10646 as well as the two are intertwined, sharing both a common history and goals. In the 1980s, text encoding experts from around the world began work on two initially parallel projects to overcome character encoding obstacles. In 1984 ISO actively started work on a universal character encoding. ISO placed heavy emphasis on compatibility with existing ISO standards, in particular ISO-8859. In the spring of 1991 ISO published a draft international standard (DIS) 10646. By that time work on Unicode was nearing completion, and many in the industry were concerned that there would be great confusion from two competing standards. In the wake of opposition to DIS-10646 from several of the ISO national bodies ISO and Unicode were asked to work together to design a common universal character code standard which came under the umbrella of Unicode. [16]

3.8.2.1 History of Unicode

The Unicode standard first began at Xerox in 1985. The Xerox team (Huan-mei Liao, Nelson Ng, Dave Opstad, and Lee Collins) was working on a database to map the relationships between the identical ideographic characters in the Japanese and Chinese character sets, and was referred to as Han unification. Around the same time Apple, and in particular Mark Davis also began development of a universal character set. [102]

In September of 1987, Joseph Becker from Xerox and Mark Davis from Apple began discussions on a universal character encoding standard for multilingual computing. In the summer of 1988 we see the first proposal for a universal character encoding, which Joseph Becker named Unicode[1]. By 1989, several people from various software companies were meeting bimonthly, creating the first full review draft of the Unicode standard. These discussions lead to the inclusion of all composite characters from the ISO-8859-x standards and Apple's Han unification work. In 1991 the Unicode consortium was officially incorporated as a nonprofit organization, and is known as Unicode Inc. [102],[103]

Urged by public pressure from various industry representatives, the ISO-10646 and Unicode design groups met in August of 1991. Together these two groups created a single universal character encoding. Naturally, compromises were made by both parties. This joint body officially published a standard in 1992, and is known as Unicode/ISO-10646. [102]

3.8.2.2 Goals of Unicode

The purpose of Unicode is to address the need for a simple and reliable world-wide text encoding. Unicode is sometimes referred to as "wide-body ASCII", due to

---

1. The first detailed description of Unicode can be found in a reprint of Joseph Becker's classic paper "Unicode 88". This reprint was published by the Unicode Consortium in 1988 in celebration of Unicode's ten year anniversary.

its use of 16 bits for encoding characters. Unicode is designed to encode all the major living languages. Unicode is a fixed width, easy to understand encoding scheme. Additionally, Unicode can support ready conversion from local or legacy encodings into Unicode, thereby easing migration. [13]

At its core Unicode can be thought of as an extension of ASCII for two reasons. First, Unicode like ASCII uses a fixed width character code. Second, Unicode like ASCII enforces a strict one-to-one correspondence with characters and code points. That is, each individual Unicode code point is an absolute and unambiguous assignment of a 16-bit integer to a distinct character. Since there are obviously more than 2^8 (256) characters in the world, the 8-bit byte in international/multilingual encodings has become insufficient. The octet equals character strategy is both too limiting and simplistic. Therefore, the 8-bit byte plays no role in Unicode. Moreover, the name Unicode was chosen to suggest an encoding that is: unique, unified, and universal. [13]

3.8.2.3 Unicode's Principles

The design of Unicode is based upon ten founding principles [13],[32],[103]:

- Fixed width encoding — In Unicode, each character is represented by a single 16-bit code point. Moreover, each character is never encoded more than once.
- Full encoding — In Unicode all code points are assigned, from 0x0000-0xFFFF; nothing is blocked out.
- Characters vs. glyphs — In Unicode a clear distinction is made between encoding characters vs. encoding glyphs. Unicode only encodes characters, which are abstract and express raw content. On the other hand, glyphs are specific visible graphic forms expressing more than content. This issue is discussed in greater detail in chapter 5.
- Semantics — In Unicode characters have properties.
- Ideographic unification — Having a clear separation between characters and glyphs permits unification of the commonly shared ideographic characters in Chinese, Japanese, and Korean.

- Plain vs. fancy text — A simple but important distinction is made between plain text which is a pure sequence of Unicode code points, and fancy text, which is any text structure that bears additional information above the pure code points. We discuss this issue in chapter 5.
- Logical order — In Unicode characters are stored in the order in which they are read, which is not necessarily the same order in which they are displayed. The concept of logical order is examined in chapter 4.
- Dynamic composition — Instead of allowing only the well known accented characters, Unicode allows dynamic composition of accented forms where any base character plus any combining character can make an accented form. We discuss dynamic composition in greater detail in chapters 4 and 5.
- Equivalent sequences — In Unicode precomposed characters are semantically equivalent to their combining counterparts. We spend considerable time discussing this throughout the rest of the dissertation.
- Convertibility — Round trip conversion between Unicode and legacy encodings is possible since each character has a unique correspondence with a sequence of one or more Unicode characters.

Naturally, some of Unicode's design goals are in direct conflict with one another. These conflicts have forced Unicode to make compromises from time to time. However, one important goal Unicode has worked hard at honoring is its ability to provide round trip conversions. In fact Joseph Becker in 1988 believed that Unicode's initial utility would be as a mechanism for interchange. This was the same reason why ASCII was constructed. Nevertheless, ASCII has transitioned from being an interchange mechanism to an outright native encoding. The same can also be said of Unicode. [13]

3.8.2.4 Differences Between Unicode and ISO-10646

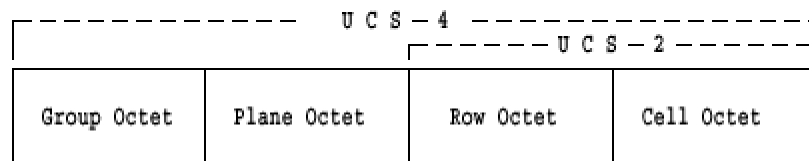
By its nature, ISO-10646 officially known as the Universal Multiple-Octet Coded Character Set, or simply known as the UCS (Universal Character Set), only describes the technical details of the UCS encoding. Additionally, Unicode includes specifications that assist implementers. Unicode defines the semantics of characters more explicitly than ISO-10646 does. For example, Unicode provides algorithms for determining the display order of Unicode text. We explore the ordering of Unicode

text streams in the next chapter. Additionally, Unicode provides tables of character attributes and conversion mappings to other character encodings. Nevertheless, Unicode and ISO-10646 are in agreement with respect to the defined characters. That is, every character encoded in Unicode is also encoded in the same position as in ISO-10646. [16]

The Unicode standard was initially designed as a 16-bit character encoding allowing for 65,535 different code points. Unicode is actually comprised of a series of planes each having 65,535 code points. If you think of the values 0x0000-0xFFFF as constituting one plane called plane 0, then you could imagine multiple planes each having 65,535 code points. Unicode refers to plane 0 as the BMP (Basic Multilingual Plane). On the other hand, ISO-10646 was designed as a 32-bit character encoding, with the most significant bit always set to 0. In ISO-1046 this 32-bit form is called UCS-4 (Universal Character Set four octet form), while the 16-bit Unicode form is called UCS-2 (Universal Character Set two octet form). [32]

Conceptually, the Universal Character Set is divided into 128 three dimensional groups. Each group contains 256 planes containing 256 rows of 256 cells. The four octets of UCS-4, therefore, represent the group, plane, row, and cell of a code point in the Universal Character Set. See Figure 3-15 [43]. A UCS-2 code point can be transformed into a UCS-4 code point by simple zero extension. [32]

Figure 3-15. Forms of UCS-4 and UCS-2



3.8.2.5 Unicode's Organization

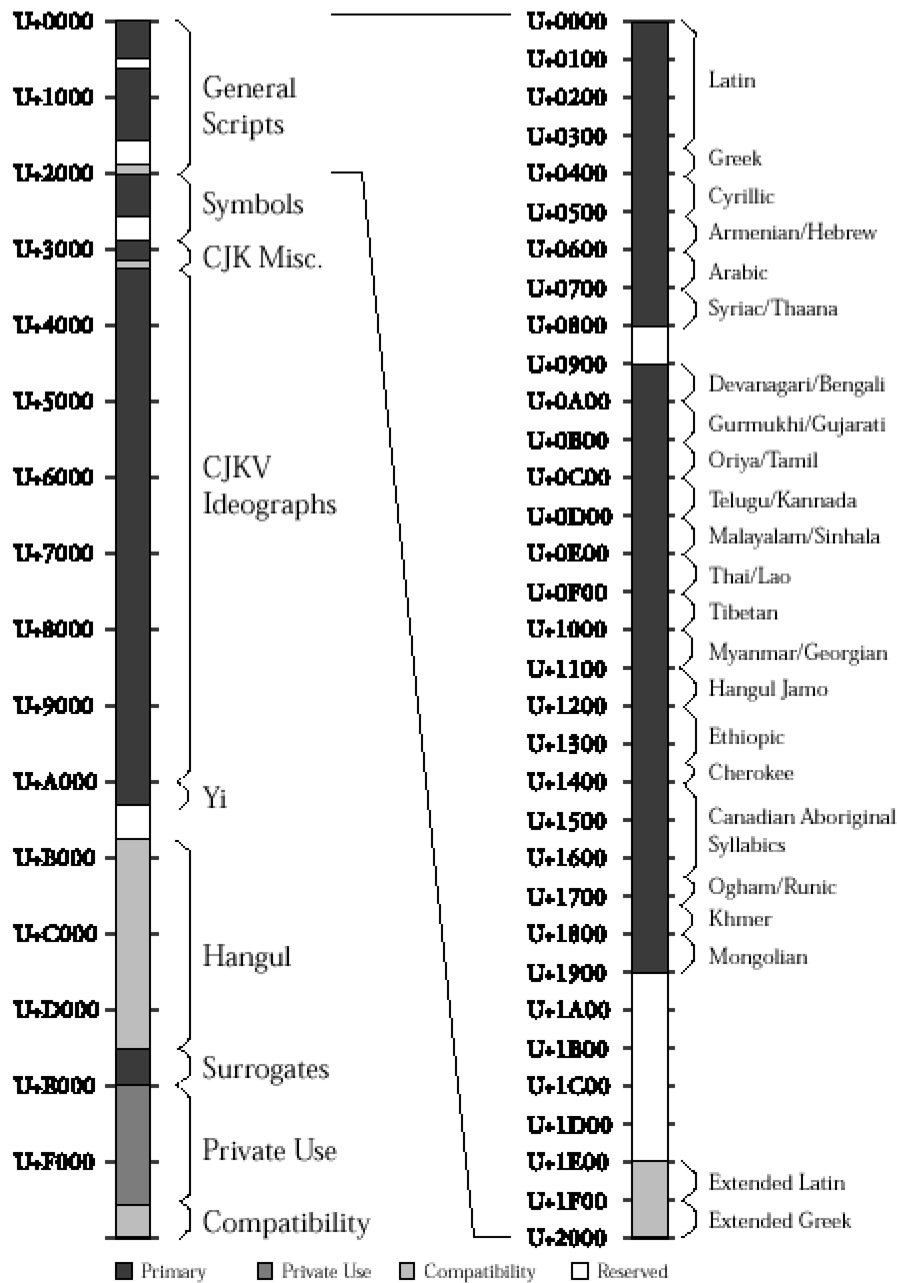
In this section we describe the layout of Unicode version 3.1, the latest version of the standard. There are two ways in Unicode to refer to code points. The first

method uses the hexadecimal value of the code point preceded by a capital letter "U" and plus sign "+". The second method is the same as the first except the plus sign is omitted. The Unicode code space is divided up according to Table 3-14. A more detailed layout is shown on Figure 3-16 [96]. Additionally, the first 256 characters within the first range of Unicode are in direct agreement with ISO-8859-1, in such a way that the 8-bit values are extended to 16-bit values by simply using zero extension. [16],[103]

Table 3-14. Unicode code point sections

Code point range (hex)	Description
U0000-U1FFF	General scripts
U2000-U2FFF	Symbols
U3000-U33FF	Chinese, Japanese, Korean miscellaneous characters
U4E00-U9FFF	Chinese, Japanese, Korean ideographs
UAC00-UD7A3	Hangul
UD800-UDFFF	surrogates
UE000-UF8FF	private use
UF900-UFFFF	compatibility and special

Figure 3-16. Unicode layout



There are three ranges of Unicode (surrogates, private use, and compatibility) that deserve special attention. First, we discuss the surrogate range. The surrogates are a range of code points that enable the extension of Unicode. The surrogate range defines 1,024 lower half and 1,024 upper half code points. The sequence of a code

point from the upper half surrogate range followed by a code point from the lower half surrogate range identifies a character in planes 1-16 according to the algorithm defined on Figure 3-17. Line 1 on Figure 3-17 takes a surrogate pair H and L (H stands for upper surrogate, L stands for lower surrogate) and returns a Unicode scalar value N. Line 2 on Figure 3-17 is the reverse of the algorithm on line 1. [32],[103],[60]

Figure 3-17. Surrogate conversion

$$N = (H - 0xD800) * 0x400 + (L - 0xDC00) + 0x10000 \quad \textbf{(1)}$$

$$H = (N - 0x10000) / 0x400 + 0xD800, L = (N - 0x10000) \% 0x400 + 0xDC00 \quad \textbf{(2)}$$

Up through Unicode version 3.0 all of Unicode's characters were defined in the BMP. However, version 3.1 of the standard is the first version that makes assignments outside the BMP. Unicode 3.1 adds 44,946 new characters and when added to the existing 49,194 characters, the new total is 94,140 characters. For the most part the new characters are additional ideographic characters that provide complete coverage of the characters in the Hong Kong supplementary character set, which was discussed earlier. [60]

The Unicode private use area is a range of Unicode that can be used by private parties for character definition. This range, for example could be used for the definition of corporate logos or trademarks. It could also be used for protocol definition when agreement is made between the interested parties. [32]

As we stated earlier Unicode was envisioned as an interchange encoding. In order to guarantee that round trip conversion would always be possible Unicode defined a special block of characters, known as the compatibility range. The compatibility range contains alternative representations of characters from existing standards. These duplicate characters are defined elsewhere within the standard. The primary purpose of these duplicates is to enable round trip mapping of Unicode and

the various national standards. Later in the dissertation we will examine several problems caused by the use of the compatibility range. [32]

3.8.2.6 Unicode Transmission Forms

There are four primary ways in which Unicode and ISO-10646 code points may be transmitted. Unfortunately, not all of the approaches are equivalent with respect to the code points that may be transmitted. The UCS-4 encoding is the only transmission mechanism that is capable of encoding all of the possible characters defined in ISO-10646. Within the semantics of Unicode, the UTF-32 (Universal Character Set Transformation Format 32-bit Form) is used to encode UCS-4. The difference being UTF-32 is restricted to the range 0x0-0x10FFFF which is precisely the range of code points defined in Unicode, while in UCS-4 all 32-bit values are valid. [32]

The UCS-2 encoding is capable of representing all of the code points defined within the BMP. Code points outside of the BMP are not represented, due to the group and plane numbers being fixed. [32]

The UTF-16 (Universal Character Set Transformation Format for Planes of Group 0) encoding permits code points defined in planes 0-16 of group 0 to be directly addressed. This is accomplished by combining individual 16-bit code points into single ISO-10646 code points using the previously mentioned algorithm on Figure 3-17. [32]

The UTF-8 (Universal Character Set Transformation 8-bit Form) encoding allows Unicode and ISO-10646 to be transmitted as a sequence of 8-bit bytes rather than as 16 or 32-bit units. It is a variable length encoding scheme requiring anywhere from one to six bytes per code point, however in the case of UTF-32 the max number of bytes would be limited to four. This is a common and useful transmission format, because UTF-8’s single-byte form directly corresponds to ASCII. Additionally, it is

safe to use in environments where code points are assumed to always be 8-bits [32],[43]. See Table 3-15. Converting from UTF-32 proceeds in three steps [110]:

- Determine the number of octets required for the character value by looking in the first column of Table 3-15.
- Prepare the high order bits of the octets as per the second through fifth columns in Table 3-15.
- Fill in the bits marked by x from the bits of the character value, starting from the low order bits putting them first in the last octet of the sequence, then next to last, and so on until all x bits are filled in.

On Table 3-16 we show how the Unicode code point U1D5A0 (Mathematical Sans-Serif Capital A) would be represented in the various Unicode transformation formats.

Table 3-15. UTF-8

UTF-32 value hex	UTF-8 1st byte	UTF-8 2nd byte	UTF-8 3rd byte	UTF-8 4th byte
0000 0000 - 0000 007F	0xxxxxxx			
0000 0080 - 0000 07FF	110xxxxx	10xxxxxx		
0000 0800 - 0000 FFFF	1110xxxx	10xxxxxx	10xxxxxx	
0001 0000 - 001F FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 3-16. Unicode transformation formats

Form	Byte sequence (hex)
UTF-32	1D5A0
UTF-16	D835,DDA0
UTF-8	F0,9D,96,A0

3.8.3 Criticism of Unicode

For the purposes of introducing the other multilingual encoding schemes we discuss some of Unicode problems.

3.8.3.1 Problems With Character/Glyph Separation

Although Unicode can remedy a large number of problems encountered by multilingual applications, it also has numerous drawbacks. Some have argued that Unicode cannot be used as a text encoding system, because of Unicode's bias towards the presentation of text. One does not have to search hard to find examples of such biases. For example, Unicode encodes the *fi* ligature as a distinct character. Most text encoding specialists would argue that *fi* is not a character, but rather is a glyph. Therefore, the *fi* glyph has no business being encoded as a character in a text encoding. In light of Unicode's orientation towards presentation, some authors have argued that Unicode should only be used as a glyph encoding. We will illustrate several other examples in later chapters, that further support this argument. [36],[74]

3.8.3.2 Problems With Han Unification

As specified, Unicode's primary purpose is to encode all the major written scripts of the world, rather than all the worlds written languages. This distinction is extremely important in Unicode. Nevertheless, most of the world's encoding systems actually encode written languages and not scripts. Recently, Unicode has provided a mechanism, known as surrogates for encoding characters that are specific to certain written languages. Nevertheless, the vast number of characters that are tied to written languages, coupled with Unicode's surrogate gymnastics hardly provide a satisfactory solution. [31]

Currently, Unicode encodes 49,194 characters in its BMP, using the Han unification process. At first, this seemed more than sufficient, however input from several nations (Japan, Mainland China, Taiwan, and Korea) was excluded during the creation of the BMP. Moreover, these were the groups that had the most characters to assign. Mainland China has responded by insisting that Unicode encode all of its official 6,000 characters in addition to the many simplified characters, plus the older classic set of some 40,000 characters. This alone would occupy nearly the entire BMP. Taiwan has also responded in a similar fashion, insisting that they have the

rights to their own complete set of classic characters. These Taiwanese characters represented an additional 50,000 characters, and would not consider using the same characters encoded by Mainland China. These two groups alone required over 90,000 distinct characters. [31]

The Japanese also said they were entitled to have their own characters encoded in a distinct range. Naturally, once Korea got wind of these requests, they also asked for their characters. If each country gets their way this could generate more than 170,000 characters. In an attempt to satisfy these groups Unicode has created surrogates. In the latest version of Unicode, 94,140 characters are encoded. This is still painfully short of the 170,000 characters needed. Obviously, 32 bits would be more than sufficient, however Unicode does not provide a 32-bit contiguous block. Clearly, two separate 16-bit blocks do not solve the problem. In order to encode the required number of characters, Unicode must resort to special encoding forms that get piggybacked onto Unicode's 16-bit form, thereby making what would be a simple problem more complex. [31]

In many cases it is necessary to know which language a stream of characters represents for data processing operations, particularly sorting, spell checking, and grammar checking. In Unicode this can be difficult to ascertain, especially if a character is in the unified Han range. This causes difficulties in creating applications for a single language, such as natural language processing. It is much easier to create these applications if all the characters come from a single language block. This furthers the argument for an encoding system that separates character blocks for different languages. [74]

3.8.3.3 ISO-8859-1 Compatibility

Unicode is not really compatible with ISO-8859-1. Unicode streams are sequences of 16-bit code points, while ISO-8859-1 streams are sequences of 8-bit code points. Unicode's encoding system does not directly recognize ISO-8859-1

data. Unicode requires that ISO-8859-1 characters be first converted to Unicode by zero extension. However, to transmit Unicode on the Internet, you have to use the 8-bit safe Unicode transformation format (UTF-8). In the case of Unicode characters that fall within in the ISO-8859-1 range the UTF-8 conversion simply removes the leading zero. [74]

3.8.3.4 Efficiency

Most data are actually in a single language, and most languages can be encoded using 8-bit code points. Using a 16-bit encoding scheme doubles both the storage requirements of programs and the transmission time of character data. Compression schemes could be used to help alleviate this, but they are impractical due to their overhead.

3.8.4 Mudawwar's Multicode

Multicode is a character encoding system proposed by Muhammad Mudawwar from the American University at Cairo in 1997 its goal is to address some of Unicode's drawbacks. Multicode's most important distinction is its use of multiple coded character sets. Multicode is not an extension to any coded character set, but rather is a collection of several coded character sets. In general, most coded character sets have strong ties to specific written languages. On the other hand, there are some characters that can be viewed as being language neutral, such as mathematical symbols. To take advantage of this approach, unlike Unicode, Multicode is oriented towards written languages and not scripts. Each coded character set used in Multicode is designed to be independent and self sufficient, each having all necessary control characters, punctuation, and special symbols. [74]

3.8.4.1 Character Sets in Multicode

Instead of attempting to merge all written languages into a single 16-bit coded character set, Multicode defines separate 8-bit and 16-bit coded character sets. In

Multicode there can be 256 separate coded character sets. Each coded character set is assigned a unique numeric identifier. In the case of ASCII the identifier is 0. [74]

In Multicode there may be substantial overlap between coded character sets, however there will be cases where a coded character set has unique characters. For example, languages based on the Latin script share many common symbols, however each has some unique letters. It is also possible to use more than one coded character set for a language. For example, the Azeri language could be written using either Cyrillic or Latin letters. [74],[111]

Multicode strives to define a unique coded character set for each written language, unlike Unicode which merges scripts through a unification processes. Additionally, Multicode supports the use of more than one coded character set standard for a given written language, in case different countries use these sets. [74]

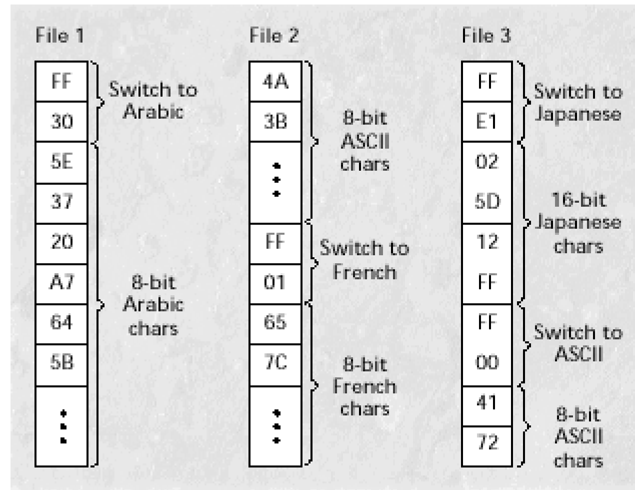
3.8.4.2 Character Set Switching in Multicode

In order to support multilingual text, Multicode provides a mechanism for switching between coded character sets. Multicode defines a special character for this purpose and that is known as a *switch character*. In every 8-bit coded character set, Multicode reserves the last code point 0xFF as the *switch character*. To switch to a different coded character set, either 8-bit or 16-bit, a special two byte sequence is inserted into the text stream. The first byte is the *switch character*, and the second byte is a *character set designator*. For example, to switch from French (assumed to be coded character set 0x01) to Hindi (assumed to be 0x50) the two byte sequence 0xFF50 would be inserted into the text stream. [74]

In each 16-bit coded character set, Multicode reserves the range 0xFF00-0xFFFF as switch characters. The first byte of the *switch sequence* is always 0xFF and represents the *switch character*. The second byte of the switch sequence is the *character set designator*. Therefore, switching in a 16-bit coded character set is really the same as switching in a 8-bit character set. See Figure 3-18. In Figure 3-18, File 1

contains a stream of Arabic characters, File 2 contains ASCII and French characters, while File 3 contains Japanese and ASCII characters. In each case a switch sequence is used to switch out of Multicode's default ASCII mode. [74]

Figure 3-18. Character set switching in Multicode



In Multicode, 16-bit coded character sets are only used in cases where a written language requires it. Multicode always uses the smallest coded character set for any given written language. When compared to Unicode, Multicode requires only half the storage for those written languages that can be represented using 8-bits. [74]

3.8.4.3 Focus on Written Languages

Multicode by design is oriented towards written languages. Each coded character is designed to encode a particular written language. Furthermore, each coded character set provides a full set of control codes, thereby eliminating the need to switch to a special character set for control functions. Additionally, language information is implicitly encoded in Multicode via the character set switch sequences. In Multicode language centric data processing is well defined because there is never any confusion over which written language a character comes from. [74]

3.8.4.4 ASCII/Unicode Compatibility

Multicode is directly compatible with ASCII. No conversion is necessary to use ASCII data in Multicode. ASCII is the default coded character set in Multicode; assigned the character set designator 0x00. Furthermore, Multicode is also compatible with Unicode. Multicode reserves the 0xFF character set designator for this purpose. There is never any misinterpretation of the switch sequence by Unicode, because 0xFFFF is an invalid Unicode character, hence it must be a switch. [74]

3.8.4.5 Glyph Association in Multicode

In Multicode several characters may share a common glyph, but have different code point values. For example, the letter *a* which appears in both the French and ASCII coded character sets could be encoded in two different positions. In Multicode characters would be associated to glyphs using either character set specific fonts or a single unified font. By using character set specific fonts, font sizes are kept to a minimum as glyphs that are unnecessary to the display of a written language are not included in the font. Additionally, there is a one-to-one mapping between characters and glyphs. On the other hand, character set specific fonts duplicate glyphs that may be common across a number of character sets. A unified font would remove this redundancy, but would require a character to glyph index conversion, because the property of a one-to-one mapping between characters and glyphs would be lost. Most notably, Unicode could be used as a unified glyph index. This would allow the use of TrueType and OpenType fonts as they use Unicode for indexing glyphs. [74]

3.8.5 TRON

TRON (The Real-Time Operating System Nucleus) is an open architecture that specifies interfaces and design guidelines for operating system kernels. The TRON Application Databus (TAD) is the standard for ensuring data compatibility across computers that support the TRON architecture. TAD supports multilingual data via multiple character sets. TAD provides both a uniform and efficient method

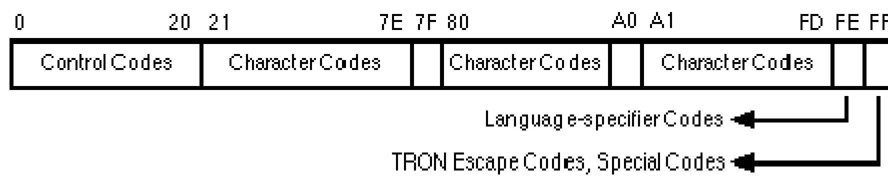
for manipulating character sets. Additionally, applications based on TAD are independent of any particular coded character set. [82]

In TAD, language specifier codes are used to switch from one language to another. Characters in TRON may be single byte, double byte or a combination of the two. At language boundaries *language specifier* codes are inserted, so that single byte and double byte codes can be intermixed within a single text stream. Therefore, TRON like Multicode always uses the most compact coded character set for a given written language. [82]

3.8.5.1 TRON Single Byte Character Code

In TRON control codes, character codes, language specifier codes, and TRON escape codes are all based on a single byte code point. See Figure 3-19. The control codes in TRON are mostly the same as ASCII's. Nevertheless, code point 0x20 (ASCII space) is treated as a control code, and is called a *separator* in TRON. The *separator* is used to indicate both word and phrase divisions as opposed to 0xA0 (*blank*). In TRON the handling of the *separator* is language specific, but in English the *separator* acts as an ASCII space. In other words, the *separator* is used as a gap when lines are broken, and it displays a variable width space for use in proportional spacing. [82]

Figure 3-19. TRON single byte character code



The character codes (0x21-0x7E, 0x80-0x9F, 0xA0, and 0xA1-0xFD) cover 220 characters. The 0xA0 character (*blank*) is handled as a fixed width space. In the case of English, the *blank* is called a *required space*. A *required space* is treated as an alphabetic character. However, the *required space* cannot be used to together with punctuation for breaking a line. [82]

The *language specifier* code 0xFE is used for switching the language of the character codes (0x21-0x7E and 0x80-0xFD). Additionally, it can be expanded into multiple bytes through repeated application of the *language specifier*. For example, the double byte sequence 0xFEFE would expand the number of language specifiers by 220. [82]

In TRON 0xFF is used as an *escape signal* when the code point that follows it is in the 0x80-0xFE range. The TRON escape is used for punctuation in text and graphic segment data. Additionally, in TRON 0xFF is used to indicate a TRON *special code* when the code point that follows it is in the 0x21-0x7E range. TRON *special codes* are used by the TRON Application Control-Flow Language and are employed as special codes that can be embedded in text. [82]

3.8.5.2 TRON Double Byte Character Code

In TRON the double byte code is divided into four character zones (A,B,C, and D), language specifier codes, TRON escape codes, and TRON special codes. See Figure 3-20. Additionally, control codes appear as single byte code points inside two byte character codes. The language specifier codes, TRON special codes, and TRON escape codes are the same as their single byte analogs. Combined, the A, B, C, and D character blocks encode 48,400 characters.

3.8.5.3 Japanese TRON Code

Japanese TRON code is a double byte code system. The A block corresponds to the JIS X0208 standard. The B block contains those frequently occurring characters that are not in JIS X0208. In the C and D blocks are infrequently used characters. The set of Latin characters that are used in Japanese are treated as belonging to the Japanese group, rather than the Latin group. Therefore, in order to mix Japanese and English, it is necessary to switch in and out of Japanese. Generally, however Latin characters are infrequently used in Japanese. When Latin characters are used, it is usually for the purpose of enumerating points in a preface using for example, the

letters A, B, C. For this reason, Latin characters are duplicated in the Japanese group. [82]

Figure 3-20. TRON double byte character code

Second Byte

First Byte	0–20	21–7E	7F	80–FD	FE	FF
0–20						
21–7E	Unused	TRON Character Codes A Zone		TRON Character Codes C Zone		
7F						
80–FD	Unused	TRON Character Codes B Zone		TRON Character Codes D Zone		
FE		Language-specifier Codes		Language-specifier Codes		
FF		Special Codes		TRON Escape Codes		

3.8.6 EPICIST

The Efficient, Programmable, and Interchangeable Code Infrastructure for Symbols and Texts (EPICIST) is a multilingual character coding system. The creators of EPICIST believe that the existing character coding standards are inflexible, insufficient, and inefficient for addressing the needs of multilingual computing. In particular, the currently available character code standards intentionally avoid the handling of private or personal characters or symbols. They only specify small ranges

of private characters. In the case of global digital libraries, which need to use non-standardized symbols, the existing approaches are woefully inadequate. A new framework is required in order to support more general or user specific symbols since formal standardization is not practical. [79]

3.8.6.1 EPICIST Code Points

EPICIST is a dynamic symbol code infrastructure for multilingual computing. EPICIST can handle both general symbols and existing defined characters. EPICIST is a variable length character coding system, which is based upon a fixed width 16-bit code point. This 16-bit code point is called an EPIC Unit (EPICU). A symbol in EPICIST consists of one or more EPICUs. The most significant bit of an EPICU is bit 16, while the least significant bit is bit 0. In an EPICU the two most significant bits are used to indicate whether the unit is the head of a symbol or a tail of a symbol. If bit 16 is 0 then the unit is the tail of a symbol. However, if bit 15 is 0 then the unit is the head of a symbol. If both bits 15 and 16 are 0, then the unit is a symbol itself. Thus, locating symbol boundaries is both simple and efficient. [79]

3.8.6.2 EPICIST Character Code Space

The code space of EPICIST is divided into subspaces. These subspaces fall into four categories: standardized character set subspaces, Epic VM (virtual machine) subspaces, user specific subspaces, and temporary subspaces. Symbol code values that consist of one or two EPICUs are predominately used for encoding the standardized characters and Epic VM instructions. Sequences of three EPICUs are reserved for future standardized characters. Symbol code values that contain four or more EPICUs are set aside for user specific symbols. [79]

3.8.6.3 Compatibility With Unicode

Just like the Unicode standard, which uses a capital *U* to indicate a Unicode code point, EPICIST uses a capital letter *P* to indicate a code point. However,

compound EPICIST symbols that are comprised of multiple EPICU's use a full-stop to delineate each unit. [79]

In EPICIST the lower code values are in direct correspondence with Unicode, except for the CJK miscellaneous symbols. For example, the Unicode character range U0000-U2FFF maps directly to the EPICIST range P0000-P2FFF. On the other hand the Unicode range U3000-U3FFF map to the EPICIST range P8000.7000-P8000.7FFF. [79]

In EPICIST combining characters are unnecessary, because every combination of combining characters can be assigned to a unique code point in EPICIST. On the other hand, Unicode must use combining characters as the code space of Unicode is insufficient if all combinations were to be defined. Therefore, Unicode uses an incomplete set of composite characters. [78]

3.8.6.4 Epic Virtual Machine

The code range P3000-P3FFF is used and set aside for Epic VM instructions and numerical representation. The code range P3E00-P3EFF contains the predefined Epic VM instructions, while the P3000-P3CFF range is marked for user defined Epic VM instructions. The code range P3F00-P3FFF is used to represent the block of integers between -128 and 127. The Epic VM decodes input symbols as instructions and executes them. Using Epic VM one can define or modify instruction definitions which may have been defined during runtime. In Epic VM a user can define a code sequence at a code point. When a symbol is input, a specified code sequence is executed. Thus, it is possible to invoke instructions as functions. [79]

3.8.6.5 Using the Epic Virtual Machine for Ancient Symbols

It is frequently difficult to standardize ancient characters that are not currently being used, but are under study by scholars. If researchers have differing opinions about the identities of symbols, then standardization is not possible. If at some point scholars can come to agreement, then ancient symbols can be standardized.

Nevertheless, academic study cannot wait for standardization. The EPICIST system allows researchers who have differing opinions about the identification of symbols to assign symbols to different code points and continue on with their investigations. Once the standardization process is complete, an Epic VM program can be embedded in EPICIST to map old code points to the new standardized ones. This is possible because an Epic VM program is nothing more than a set of symbols and are transmitted along with data encoded in EPICIST. [78]

3.8.7 Current Direction of Multilingual Encodings

It appears Unicode is the prominent multilingual encoding. Some of the reasons for this are based on sound technical arguments, while others are for political and or commercial reasons. Technically, working with Unicode is actually no more difficult than working with ASCII, because of Unicode's fixed width stateless characters. On the other hand, Multicode, TRON, and EPICIST require either the manipulation of multi-byte characters, the manipulation of variable length code sequences, or maintaining stateful information.

Commercially, Unicode has been a major success. Unicode can be found in operating systems (Linux, MacOS, OS/2, and Windows), programming languages (Java, Perl, and Python) as well as web browsers (Mozilla, Netscape, and Internet Explorer). Therefore, we use Unicode as a basis for illustrating information processing problems that arise from adopting a multilingual encoding.

4 Bidirectional Text

Unicode's ability to mix the various script systems of the world makes the creation of multilingual documents no more difficult than the creation of monolingual documents. But this causes difficulties. An example of text using two different script systems is given in Figure 4-1. This text is an excerpt from a Tunisian newspaper, and tells of an upcoming international music festival. In this example we see an English phrase "(Tabarka World Music Festival)" embedded in a paragraph that is comprised of mostly Arabic text. The paragraph also contains European numerals for the date. The beginning of the paragraph starts in the upper right hand corner, and is read from right-to-left except when numerals or English phrases are encountered. We call such text streams "bidirectional text". [4]

Figure 4-1. Tunisian newspaper

بعد الإقبال الجماهيري المكثف والمنقطع النظير لفعاليات الدورة السادسة لمهرجان
طبرقة الدولي للجاز (من 29 جوان الى 7 جويلية الجاري) متجاوزا بذلك توقعات
المنظمين بالديوان الوطني التونسي للسياحة ... ستشهد مدينة المرجان تنظيم تظاهرة
موسيقية دولية أخرى خلال الأسبوع الأخير لشهر أوت القادم تحت عنوان : مهرجان
طبرقة للوورلد ميوزك» (Tabarka World Music Festival) وكما تدل عليه
التسمية فإنه من المنتظر أن يستقطب كبرى المجموعات الموسيقية العالمية في نمط ما
يعرف بـ«الوورلد ميوزك» (موسيقى العالم) والموسيقى الإتنية القادمة من مختلف بقاع
العالم والمتشبعة بتراث الشعوب وفنونها .

For the most part the layout of such bidirectional paragraphs is fairly straight forward. There are subtleties however, that can make the layout become non-trivial. Additionally, in some cases ambiguities may arise from the intermixing of script systems with conflicting directions. The goal of this chapter is to explore some of these

subtleties and ambiguities. In particular, great attention is given to the intermixing of Latin based scripts (written left-to-right) with the Arabic and Hebrew script systems (written right-to-left). We demonstrate that the layout of multilingual text is non-trivial. This is followed by an investigation of the current techniques (algorithms) that are being used for layout of multilingual text. Lastly, the deficiencies in the current strategies are illustrated.

4.1 Non Latin Scripts

The inexact match between phoneme (phonetic unit that represents a distinct sound in a language) and orthographic representation has made it possible for English to represent its intricate system of sounds with out the need of diacritical marks (modifying marks that alter the phonetic value of a character). Each word in English can be in encoded in ASCII. The remaining Latin script languages rely strongly on the use of diacritical marks and hence cannot be correctly encoded in ASCII. [62]

The addition of diacritical marks to an alphabet cannot help but complicate layout and editing. In some scripts the actual glyphs (visual shape of a character or a sequence of characters) are altered dramatically. The reason for this lies in the history of literacy in the language. The glyphs for a set of alphabetic characters is strongly connected to the medium on (or in) which the glyphs are rendered. For example, the graphic shapes representing the syllabary of Sumerian were created by pressing a narrow triangular shaped stylus into clay, producing wedge shaped marks, known as cuneiform, from which the script gets its name. [62]

The more recent Semitic scripts, of which Arabic is presently the most widespread, are pen and ink scripts. The development of Arabic as an efficient handwriting has made it relatively hard to work with in an automated environment. This difficulty comes not only from Arabic's cursive nature, but also from its bidirectional (an intermixing of text segments written right-to-left with segments written left-to-right) layout requirements. These challenges are discussed in later sections. [62]

4.1.1 Arabic and Hebrew Scripts

Arabic writing is alphabetical. Ideally alphabets consist of a few dozen letters, each of them representing only one unique sound. In modern Arabic there are 28 basic letters, 8 of them doublets differentiated by diacritics and 6 optional letters for representing vowels. The letters are written from right-to-left, with words being separated by white space. The letters within a word are generally connected to each other. Numerals are read from left-to-right just like the Latin based languages. From a strictly information processing perspective this is quite similar to Latin based scripts, disregarding the right-to-left writing direction and the interconnecting of letters. [71]

When the first attempts were made to construct a type font for Arabic, there was no model from which to construct glyphs other than handwriting. The Arabic language was not often inscribed on stone, so stonecutters were not given any incentive to create their own glyphs in spite of the popularity of stone monuments and inscriptions. Nevertheless, Arabic is difficult to capture in computers because Arabic is a hand written script requiring some amount of compromise for discrete characters. The compromise of using discrete characters to codify Arabic writing makes it difficult to express certain intrinsic properties (cursive, position, ligatures, and mirrors) of the Arabic script. We examine these properties below. [62]

4.1.1.1 Cursive

The finest Arabic inscriptions are imitations of handwriting, and are almost always cut in relief. A calligrapher would paint an inscription on a surface from which a stonecutter would then chisel away the unpainted stone. This left the letters standing out against a background. Nevertheless, this fluid, connected nature of Arabic is difficult to adapt to the technology of movable type or matrix based glyph design. [62]

4.1.1.2 Position

In Arabic, and to some extent in Hebrew, the mapping of a glyph to a character is not one-to-one as in the Latin script. Instead the selection of a character's glyph is based upon its position within a word. Subsequently, each Arabic character may have up to four possible shapes: [6], [21], [89]

- Initial - Character appears in the beginning of a word
- Final - Character appears at the end of a word.
- Medial - Character appears somewhere in the middle.
- Isolated - Character is surrounded by white space.

Furthermore, glyph selection must also take into consideration the linking abilities of the surrounding characters. For example, some glyphs may only link on their right side while others may permit links on either side. In Arabic each character belongs to one of the following joining classes:[96]

- Right joining - Alef, Dal, Thal, Zain
- Left joining - None
- Dual joining - Beh, Teh, Theh, ...
- Join causing - Tatweel, Joiner (U200D)
- Non joining - Spacing characters, Non-joiner (U200C)
- Transparent - Combining marks

In Hebrew some characters do have final forms even though Hebrew is not a cursive script. The idea of contextual shaping is certainly not limited just to right-to-left scripts. For example, the Greek script provides a special final form for the sigma character. [96]

4.1.1.3 Ligatures

Occasionally two or more glyphs combine to form a new single glyph called a ligature. This resultant shape then replaces the individual glyphs from which it is comprised. In Arabic this occurs frequently and in Hebrew rarely. In particular the

Alef Lamed ligature (UFB4F) is used in liturgical books. The number of actual ligatures used in Arabic text is difficult to determine. However Unicode devotes nearly 1,000 code points for them. Although infrequent, ligatures do occur even in English. Specifically, the fi ligature where the letter f merges with the letter i. [6], [21], [89]

4.1.1.4 Mirroring

In some cases glyph selection may be based on a character's direction. These characters are known as mirrored characters (parentheses and brackets). When mirrored characters are intermixed with Arabic and or Hebrew characters, a complementary shape may need to be selected so as to preserve the correct meaning of an expression. For example, consider the text stream $1 < 2$ in logical order (one less than two). If this stream is to be displayed in a right-to-left sequence it must be displayed as $2 > 1$. In order to preserve the correct meaning the $<$ is changed to $>$. This process is known as mirroring or symmetric swapping. [89]

4.1.2 Mongolian Script

Mongolian writing is also alphabetic, like Arabic. In Mongolian there are 27 basic letters, and 8 letters for representing vowels. Words are separated by white space. Mongolian's ancestor, classic Uigur script belonged to the right-to-left Arabic script family. Like other Arabic based scripts, a character's shape is based upon its position within a word. This makes Mongolian and Arabic quite similar, however Mongolian has more complicated orthographies. In some cases position information is not always enough to specify final glyphs, and there can even be some variation for the same form. Under Chinese influence Mongolian is now written vertically in columns from top to bottom, in a general left-to-right direction. Nevertheless, this script system brings yet another challenge to information processing. [53]

4.2 Bidirectional Layout

As computing power increases and as high quality laser printers become commonplace, user expectations rise. The computer must now be able to take sequences of intermixed characters (left-to-right and right-to-left) and place them in their proper position. We call this process “bidirectional layout”. In this section we explore some of the issues related to bidirectional layout.

4.2.1 Logical and Display Order

For the most part the order in which characters are stored in typesetting systems (logical order) is equivalent to the order in which they are visually presented (display order). The only exceptions being those scripts that are written from right-to-left. When the logical and display orders are not equivalent an algorithm is required to convert the logical order to display order. At first this might seem trivial, given that a right-to-left script simply has its display order in reverse. Unfortunately this is not the case. Technically, Arabic and Hebrew are not simply right-to-left scripts, rather they are bidirectional scripts. This bidirectional nature is exhibited when alphabetic and numeric data are intermixed. For example, the digits 2 and 9 in Figure 4-1 are the number 29 and not 92. Therefore, an algorithm that simply reverses characters is inadequate. [10]

Additionally, we must also consider text data that is comprised from various script systems. As soon as any word or phrase from a non right-to-left script (English, German, etc.) is incorporated into a right-to-left script (Arabic, Hebrew, etc.), the same bidirectional problem arises. In certain cases the correct layout of a text stream may be ambiguous even when the directions of the scripts are known. Consider the following example in Figure 4-2 in which Arabic letters are represented by upper case Latin characters.

Figure 4-2. Ambiguous layout

fred does not believe TAHT YAS SYAWLA I

In the absence of context (a base or paragraph direction) there are two possible ways to read the sentence. When read from left-to-right (Fred does not believe I always say that), and when read from right-to-left (I always say that Fred does not believe.) It thus becomes apparent that the problem is not only an algorithmic one but a contextual one. [41]

4.2.2 Contextual Problems

A logical to display conversion algorithm must also contend with the problem of context. In many cases an algorithm must consider the context in which a sequence of characters (alphabetic and numeric) appears. This can lead to cases in which an algorithm will yield inappropriate results when the context is not known or misunderstood.

Consider a phone number appearing in a stream of Arabic letters, MY NUMBER IS (321)713-0261. In this example uppercase Latin letters represent Arabic letters, and the digits represent European numerals. This should not be rendered as a mathematical expression. In Arabic mathematical expressions are read right-to-left, while phone numbers are read left-to-right. See Figure 4-3. [12],[96]

Figure 4-3. Rendering numbers

0261-713(321) SI REBMUN YM **(incorrect)**

(321)713-0261 SI REBMUN YM **(correct)**

Without understanding the context in which numbers appear, the correct display cannot be determined. There are numerous contextual and cultural factors (e.g., language and locale) that need to be given consideration when converting to display order.

4.2.3 Domain Names

In some situations a character changes meaning based on context. Consider the use of the hyphen-minus character in domain names. In domain names the predominant usage of the hyphen-minus is as white space and not as a mathematical operator or sign indicator. The example in Figure 4-4 illustrates the effect of European digits surrounding the hyphen-minus characters.

Line 1 on Figure 4-4 is a single domain name label in logical order. In this example uppercase Latin letters represent Hebrew letters, and the digits represent European numerals. Line 2 is the same label in display order, this is the output if the hyphen-minus characters are treated as mathematical operators. The text on Line 3 is also in display order, however this output is obtained when the hyphen-minus characters are treated as white space characters.

Figure 4-4. Using a hyphen minus in a domain name

NOP--123 **(1)**

--123PON **(2)**

123--PON **(3)**

Exploring domain names further, we see that even the full-stop character's semantics change based on context. The text on Line 1 of Figure 4-5 is a domain name in logical order, uppercase Latin letters represent Arabic letters. Line 2 is the resultant display order if the full-stop is treated as a sentence terminator (punctuation). In this example the presence of an Arabic character in the first label forces the entire domain name to take on an overall right-to-left reading. This is certainly correct behavior if this is the first sentence in a paragraph, however this is inappropriate in the context of a domain name. This behavior unfortunately mangles the hierarchical structure of the domain name. We suggest that the output on line 3 is more desir-

able, as this output is consistent with the current structure of domain names. In this case the full-stop characters are ignored.

Figure 4-5. Using a full-stop in a domain name

ABC.ibm.com	**(1)**
ibm.com.CBA	**(2)**
CBA.ibm.com	**(3)**

4.2.4 External Interactions

The layout of bidirectional text is a complex process requiring the interaction of various systems. We discussed above the contextual problem in bidirectional layout and how it is solved by contextual analysis and character reordering. This is only one piece of the puzzle.

4.2.4.1 Line Breaking

When bidirectional text is displayed or printed it is done so on a line by line basis for each paragraph. The lines, however are not actually comprised of characters, but rather glyphs. The process of constructing the lines, “line breaking”, requires that the widths of all the glyphs in the paragraph along with the width of the display area be known. It is inappropriate to assume that the number of and width of each character is the same when displayed. This requires a sophisticated mapping between characters and glyphs. [6], [21]

4.2.4.2 Glyph Mapping

Traditionally glyphs are selected and drawn by font rendering engines rather than via character replacement. The logic for this approach is centered around glyph availability. Some glyphs may simply not be available in a font (e.g., Hebrew and Greek final forms). If implementers were to replace sequences of characters with new character ligatures, there would be no guarantee that they would be present in a font as well. Some ligatures are not able to be constructed by using character replacement,

as they are not present in Unicode. The choice of an appropriate glyph requires knowledge of the font and its available glyphs.

4.2.4.3 Behavioral Overrides

Putting aside glyph related problems there are still other facets in a complete layout solution. For example, user supplied information may be required in order to determine where a paragraph begins and ends. Examining just the stream contents isn't always sufficient. This information could appear as control codes or be supplied externally. [96]

In some cases user preferences or locales can also force the stream contents to change. For example, the shapes used to display numeric characters could be controlled by a locale. In an Arabic locale numeric characters would be displayed with "Hindi" shapes, while a Western European locale would use "Arabic" shapes for numbers. [41]

4.2.5 Bidirectional Editing

There are also aspects of bidirectional layout that are outside the scope of overrides, in particular the caret and the mouse. Movement of the caret and hit testing of the mouse becomes more complex in bidirectional streams. If the caret is moving linearly within one of the (logical or visual) streams, then this movement needs to be translated to the other stream. Highlighting poses a similar problem as to which stream is being highlighted (logical or visual). [6], [21]

4.2.6 Goals

Unfortunately, the tasks that the developer would like to provide are not necessarily the same ones that can be provided. All of this depends on how the algorithm is intended to be used. If the intended use is to fit within in some broader context then it may be acceptable to leave some features out. If the intended use is to provide a complete layout framework, a set of features above and beyond the ones mentioned

may be required. The specification of a bidirectional algorithm can only be implemented as a character stream reordering (What else can an implementer do?), yet the bidirectional layout problem can only be solved in a larger context.

4.3 General Solutions to Bidirectional Layout

There are four general ways in which the bidirectional display problem can be addressed. Three of these strategies are automated, while one requires user intervention:

- Forced Display
- Explicit
- Implicit
- Implicit/Explicit

4.3.1 Forced Display

The Forced Display algorithm requires users to enter characters in display order. So if a text stream contained Arabic (right-to-left) characters the user would simply enter them backwards. This inelegant solution becomes cumbersome when scripts are intermixed. On the other hand, this approach has the advantage that the output (display order) is always correct and independent of the context. [12]

4.3.2 Explicit

Another potential solution to the bidirectional problem is to allow users to enter text in logical order but expect them to use some explicit formatting codes (for example, U202B and U202A in Unicode) for segments of text that run contrary to the base text direction, but what does one do with the explicit control codes in tasks other than display? For example, what effect should these controls have on searching and data interchange. These explicit codes require specific code points to be set-aside for them as well. In some encodings this may be unacceptable due to the fixed number of code points available and the number of code points required to represent the script

itself. A less technical problem is the pain this process causes the users, requiring constant thought in terms of presentation, which is an unnatural way to think about text. [12], [41]

4.3.3 Implicit

Humans want to be able to enter text in the same way as one would read it aloud. Ideally, one would like to maintain the flexibility of entering characters in logical order while still achieving the correct visual appearance. Such "implicit layout algorithms" do exist. They require no explicit directional codes nor any higher-order protocols. These algorithms can automatically determine the correct visual layout by simply examining the logical text stream. Generally the implicit rules are sufficient for the layout of most text streams. Still, there are situations in which an implicit algorithm will not always yield an acceptable result, because it is difficult to design a set of heuristics for every situation. [41]

4.3.4 Implicit/Explicit

An implicit/explicit Algorithm offers the greatest level of flexibility by providing a mechanism for unambiguously determining the visual representation of all raw streams of text. This type of algorithm combines the benefits of implicit layout algorithms with the flexibility of an explicit algorithm. Throughout the rest of this chapter we limit our discussion of bidirectional algorithms to this type of algorithm, because it shows the greatest potential for solving the bidirectional display problem. [96]

4.4 Implicit/Explicit Bidirectional Algorithms

The primary algorithm explored below is the Unicode Bidirectional Algorithm. This algorithm is in the implicit/explicit class of bidirectional algorithms. The other algorithms that are discussed in this section are variations of Unicode's algorithm.

4.4.1 Unicode Bidirectional Algorithm

The Unicode Bidirectional Algorithm is described in Unicode Technical Report #9. There are two reference implementations — one written in the programming language Java and one in C [100]. The Unicode algorithm is based upon existing implicit layout algorithms and explicit directional control codes that may be in the input stream.

The core of the Unicode Bidirectional algorithm is centered around three aspects: resolving character types, reordering characters and analyzing mirrors. The bidirectional algorithm is applied to each paragraph on a line by line basis. During resolution, characters that do not have a strong direction are assigned a direction based on the surrounding characters or directional overrides. In this context the term "strong" indicates a character that is either a left-to-right character or a right-to-left character. In the reordering phase, sequences of characters are reversed as necessary to obtain the correct visual ordering. Finally, each mirrored character (parenthesis, brackets, braces, etc.) is examined to see if it needs to be replaced with its symmetric mirror.[100]

The Unicode Bidirectional Algorithm determines the general reading direction of a paragraph either explicitly or implicitly. In the explicit method the reading direction of a paragraph is communicated to the algorithm outside of and independent from the characters in the paragraph. The implicit method determines the reading direction of a paragraph by applying a set of heuristics on the characters in the paragraph. [100]

4.4.2 IBM Classes for Unicode (ICU) and Java

Java 1.2 provides a complete framework for creating multi script applications. Java's TextLayout and LineBreakMeasurer classes facilitate the layout of complex text in a platform neutral manner. The underlying approach to reordering is based on the Unicode Bidirectional Algorithm. [21]

ICU's approach is very close to Java due in some respect to the fact that the overall internationalization architecture of Java is based on ICU. The key differences are centered around glyph management. In ICU glyph management routines are not necessary because ICU is not designed to be a complete programming environment. The ICU components are designed to work in conjunction with other libraries. [45]

4.4.3 Pretty Good Bidirectional Algorithm (PGBA)

Mark Leisher's PGBA is another algorithm for bidirectional reordering. The algorithm takes an implicit approach to reordering. PGBA does not attempt to match Unicode's reordering algorithm. However PGBA's implicit algorithm does match the implicit section of the Unicode Bidirectional Algorithm. At the moment it does not support the explicit bidirectional control codes (LRE, LRO, RLE, RLO, PDF). One should not infer that the lack of support for directional control codes results in an incomplete algorithm. Under most circumstances the implicit algorithm reorders a text stream correctly. Secondly, these control codes are not always present in all encoding schemes. Of course it would be a nice feature, but certainly not a necessary one. [56]

4.4.4 Free Implementation of the Bidirectional Algorithm (FriBidi)

Dov Grobgeld's FriBidi follows the Unicode Bidirectional Reference more closely. Notably there is support for integration with graphical user interfaces along with a collection of code page converters. However as in PGBA the explicit control codes are not currently supported. [34]

4.5 Evaluation of Bidirectional Layout Algorithms

In this section we report on the results of our independent evaluation of the output of the bidirectional algorithms. The primary goal we sought in evaluating the algorithms was to determine whether or not their output matched Unicode's reference

algorithm. We have tested them on a large number of small, carefully crafted test cases of basic bidirectional text.

4.5.1 Testing Convention

To simulate Arabic and Hebrew input/output a simple set of rules are utilized. These rules make use of characters from the Latin 1 charset. The character mappings allow Latin 1 text to be used instead of real Unicode characters for Arabic, Hebrew, and control codes. This is an enormous convenience in writing, reading, running and printing the test cases. This form is the same as the one used by the Unicode Bidirectional Reference Java Implementation [100]. See Table 4-1. Unfortunately not all of the implementations adhere to these rules in their test cases. To compensate for this, changes were made to some of the implementations.

Table 4-1. Bidirectional character mappings for testing

Type	Arabic	Hebrew	Mixed	English
L	a - z	a - z	a - z	a - z
AL	A - Z		A - M	
R		A - Z	N - Z	
AN	0 - 9		5 - 9	
EN		0 - 9	0 - 4	0 - 9
LRE	[	[	[	[
LRO	{	{	{	{
RLE	]	]	]	]
RLO	}	}	}	}
PDF	^	^	^	^
NSM	~	~	~	~

In the Unicode C reference implementation additional character mapping tables were added to match those of the Unicode Java Reference implementation. Also the bidirectional control codes were remapped from the control range 0x00-

0x1F to the printable range 0x20-0x7E. This remapping allows test results to be compared more equitably.

In PGBA and FriBidi the character attribute tables were modified to match the character mappings outlined in Table 4-1. The strategy for testing ICU and Java was slightly different than PGBA and FriBidi. In the ICU and Java test cases we used the character types rather than character mappings.

4.5.2 Test Cases

The test cases are presented in Tables: 4-2, 4-3, 4-4, and 4-5. The source column of each table shows the test input. The expected column is what we think the correct output should be. In all cases this is the output produced by our HaBi implementation. These test cases are taken from the following sources:

- Mark Leisher - His web page provides a suite of test cases as well as a table of results for other implementations [56]. See Tables: 4-2 and 4-3.
- Unicode Technical Report #9 - Some of the examples are used for testing conformance [100]. See Table 4-2.
- Additional test cases for uncovering potential bugs in an implementation's handling of weak types and directional controls. See Tables: 4-4 and 4-5.

Table 4-2. Arabic charmap tests

	Source	**Expected**
1	car is THE CAR in arabic	car is RAC EHT in arabic
2	CAR IS the car IN ENGLISH	HSILGNE NI the car SI RAC
3	he said “IT IS 123, 456, OK”	he said “KO ,456 ,123 SI TI”
4	he said “IT IS (123, 456), OK”	he said “KO ,(456 ,123) SI TI”
5	he said “IT IS 123,456, OK”	he said “KO ,123,456 SI TI”
6	he said “IT IS (123,456), OK”	he said “KO ,(123,456) SI TI”
7	HE SAID “it is 123, 456, ok”	“ok ,456 ,123 it is” DIAS EH
8	<H123>shalom</H123>	<123H/>shalom<123H>
9	HE SAID “it is a car!” AND RAN	NAR DNA “!it is a car” DIAS EH
10	HE SAID “it is a car!x” AND RAN	NAR DNA “it is a car!x” DIAS EH

Table 4-2. Arabic charmap tests (Continued)

	Source	Expected
11	-2 CELSIUS IS COLD	DLOC SI SUISLEC 2-
12	SOLVE 1*5 1-5 1/5 1+5	5+1 5/1 5-1 5*1 EVLOS
13	THE RANGE IS 2.5..5	5..2.5 SI EGNAR EHT
14	IOU $10	10$ UOI
15	CHANGE -10%	%10- EGNAHC
16	-10% CHANGE	EGNAHC %10-
17	he said “IT IS A CAR!”	he said “RAC A SI TI!”
18	he said “IT IS A CAR!X”	he said “X!RAC A SI TI”
19	(TEST) abc	abc (TSET)
20	abc (TEST)	abc (TSET)
21	#@$ TEST	TSET $@#
22	TEST 23 ONCE abc	abc ECNO 23 TSET
23	he said “THE VALUES ARE 123, 456, 789, OK”	he said “KO ,789 ,456 ,123 ERA SEULAV EHT”.
24	he said “IT IS A bmw 500, OK.”	he said “A SI TI bmw KO ,500.”

Table 4-3. Hebrew charmap tests

	Source	Expected
1	HE SAID “it is 123, 456, ok”.	.”it is 123, 456, ok” DIAS EH
2	<H123>shalom</H123>	<123H/>shalom<123H>
3	<h123>SAALAM</h123>	<h123>MALAAS</h123>
4	-2 CELSIUS IS COLD	DLOC SI SUISLEC -2
5	-10% CHANGE	EGNAHC -10%
6	TEST ~~~23%%% ONCE abc	abc ECNO 23%%%~~~ TSET
7	TEST abc ~~~23%%% ONCE abc	abc ECNO abc ~~~23%%% TSET
8	TEST abc@23@cde ONCE	ECNO abc@23@cde TSET
9	TEST abc 23 cde ONCE	ECNO abc 23 cde TSET
10	TEST abc 23 ONCE cde	cde ECNO abc 23 TSET
11	Xa 2 Z	Z a 2X

Table 4-4. Mixed charmap tests

	Source	**Expected**
1	A~~	~~A
2	A~a~	a~~A
3	A1	1A
4	A 1	1 A
5	A~1	1~A
6	1	1
7	a 1	a 1
8	N 1	1 N
9	A~~ 1	1 ~~A
10	A~a1	a1~A
11	N1	1N
12	a1	a1
13	A~N1	1N~A
14	NOa1	a1ON
15	1/2	1/2
16	1,2	1,2
17	5,6	5,6
18	A1/2	2/1A
19	A1,5	1,5A
20	A1,2	1,2A
21	1,.2	1,.2
22	1,A2	2A,1
23	A5,1	5,1A
24	+$1	+$1
25	1+$	1+$
26	5+1	5+1
27	A+$1	1$+A
28	A1+$	$+1A
29	1+/2	1+/2
30	5+	5+

Table 4-4. Mixed charmap tests (Continued)

	Source	Expected
31	+$	+$
32	N+$1	+$1N
33	+12$	+12$
34	a/1	a/1
35	1,5	1,5
36	+5	+5

Table 4-5. Explicit override tests

	Source	Expected
1	a}}}def	afed
2	a}}}DEF	aFED
3	a}}}defDEF	aFEDfed
4	a}}}DEFdef	afedFED
5	a{{{def	adef
6	a{{{DEF	aDEF
7	a{{{defDEF	adefDEF
8	a{{{DEFdef	aDEFdef
9	A}}}def	fedA
10	A}}}DEF	FEDA
11	A}}}defDEF	FEDfedA
12	A}}}DEFdef	fedFEDA
13	A{{{def	defA
14	A{{{DEF	DEFA
15	A{{{defDEF	defDEFA
16	A{{{DEFdef	DEFdefA
17	^^abc	abc
18	^^}abc	cba
19	}^abc	abc
20	^}^abc	abc
21	}^}abc	cba
22	}^{abc	abc

Table 4-5. Explicit override tests (Continued)

	Source	Expected
23	}^^}abc	cba
24	}}abcDEF	FEDcba

4.5.3 Test Results

All implementations were tested by using the test cases from Tables: 4-2, 4-3, and 4-4. The implementations that support the Unicode directional control codes (LRO, LRE, RLO, RLE, and PDF) were further tested using the test cases from Table 4-5. At this time, the directional control codes are only supported by HaBi, ICU, Java 1.2, Unicode Java reference, and Unicode C reference.

When the results of the test cases were compared, the placement of directional control codes and choice of mirrors was ignored. This is permitted by Unicode since the final placement of control codes is arbitrary, and mirroring may optionally be handled by a higher-order protocol.

Table 4-6. Arabic test differences

	PGBA 2.4	FriBidi 1.12
2		SI RAC the car NI ENGLISH
4	he said “KO ,)456 ,123(SI TI”	
6	he said “KO ,)123,456(SI TI”	
7		DIAS EH “it is 456 ,123, ok”
8		<123H>shalom</123H>
9		DIAS EH “it is a car!” DNA RAN
10		DIAS EH “it is a car!x” DNA RAN
11		-SI SUISLEC 2 COLD
12	1+5 1/5 1-5 5*1 EVLOS	
14	$10 UOI	
15	%-10 EGNAHC	10- EGNAHC%
16	EGNAHC %-10	-10% CHANGE
19	abc)TSET(	(TSET) abc

Table 4-6. Arabic test differences (Continued)

	PGBA 2.4	FriBidi 1.12
21		#@$ TEST
22		ECNO 23 TSET abc
24	he said “A SI TI bmw 500, KO.”	

Table 4-7. Hebrew test differences

	PGBA 2.4	FriBidi 1.12
5	EGNAHC %-10	
6	abc ECNO %%%23~~~ TSET	
7	abc ECNO %%%23~~~ abc TSET	
11	Z 2 aX	a 2X

Table 4-8. Mixed test differences

	PGBA 2.4	FriBidi 1.12
1		A~~
2	~a~A	~Aa~
10	1a~A	~Aa1
14	1aON	
18	1/2A	1/2A
19		5,1A
21		2.,1
23		1,5A
27		+$1A
28		1+$A
32	1$+N	
35		5,1

Tables: 4-6, 4-7, and 4-8 detail the differences among the implementations with respect to the results obtained with the HaBi Implementation. Only PGBA and FriBidi return results that are different than the HaBi implementation. The Unicode Java reference, Unicode C reference, Java 1.2, and ICU pass all test cases.

In PGBA, types AL and R are treated as being equivalent [56]. This in itself does not present a problem as long as the data stream is free of AL and EN (European number). However, a problem arises when an AL is followed by an EN for example, test case 18 from Table 4-4. In this situation the ENs should be treated as ANs (Arabic number) and not left as ENs.

The handling of a NSM is also different in PGBA. PGBA treats a NSM as being equal to an ON (other neutral) [56]. This delays the handling of NSM until the neutral type resolution phase rather than in the weak type resolution phase. By delaying their handling, the wrong set of rules are used to resolve the NSM type. For example, in test case 2 from Table 4-4 the last NSM should be treated as type L instead of type R.

In FriBidi there are a few bugs in the implementation. Specifically, when an AL is followed by an EN the EN is not being changed to type AN. See test case 18 in Table 4-4. This is the same symptom as was found in PGBA, but the root cause is different. In FriBidi, step W2 (weak processing phase rule two) the wrong type is being examined it should be type EN instead of type N.

There is also a bug in determining the first strong directional character. The only types that are recognized as having a strong direction are types R and L. Type AL should also be recognized as a strong directional character. For example, when test case 1 from Table 4-4 is examined FriBidi incorrectly determines that there are no strong directional characters present. It then proceeds to default the base direction to type L when it should actually be of type R. This problem also causes test cases 2, 9, and 11 from Table 4-2 to fail.

4.6 Functional Approach to Bidirectional Layout

Having examined several bidirectional algorithms we see numerous problems with the implementations. We believe these errors are not the fault of the implemen-

tations, but rather are the fault of the algorithm and its description. In this section we introduce the Haskell bidirectional algorithm (HaBi). Our goal is to apply functional programming techniques to the problem of bidirectional layout, so as to discover the essence of the Unicode Bidirectional Algorithm. A greater understanding of the algorithm is obtained by a clear functional description of its operations [41]. Without a clear description, implementers may encounter ambiguities that ultimately lead to divergent implementations, contrary to the primary goal of the Unicode Bidirectional Algorithm. During the construction of our functional implementation we excluded all references to the Java and C implementations of the Unicode Bidirectional Algorithm, so as to prevent any bias.

4.6.1 Haskell Bidirectional Algorithm (HaBi)

In this section the source code to HaBi is presented. The HaBi reference implementation uses the Hugs 98 version of Haskell 98 [51] as it is widely available (Linux, Windows, and Macintosh) and easily configurable.

Since the dominant concern in HaBi is comprehension and readability our implementation closely follows the textual description as published in the Unicode Technical Report #9. See Figure 4-7. HaBi is comprised of five phases as in the Java Unicode Bidirectional Reference implementation:

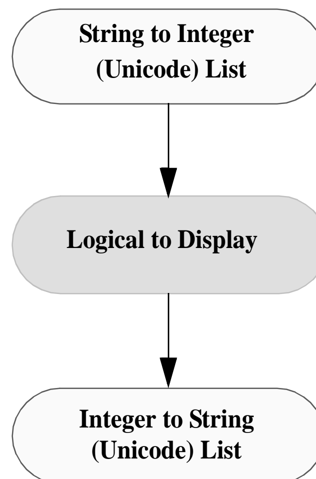
- Resolution of explicit directional controls
- Resolution of weak types
- Resolution of neutral types
- Resolution of implicit levels
- Reordering of levels

Currently there is no direct support for Unicode in the Hugs 98 implementation of Haskell 98[1]. So we treat Unicode as lists of 16 or 32-bit integers. The authors provide two modules for Unicode manipulation. The first is used to create Unicode (UCS4, UCS2, and UTF-8) strings. The second is used for determining character

types. Utility functions convert Haskell strings with optional Unicode character escapes to 16 or 32-bit integer lists. A Unicode escape takes the form \u*hhhh* analogous to Java. This escape sequence is used for representing code points outside the range 0x00-0x7F. This format was chosen so as to permit easy comparison of results to other implementations.

Internally HaBi manipulates Unicode as sequences of 32-bit integers. See Figure 4-6. HaBi is prepared to handle surrogates as soon as Unicode assigns them. The only change HaBi requires is an updated character attribute table. It would be more elegant to use the polymorphism of Haskell since the algorithm does not really care about the type of a character only its attribute.

Figure 4-6. Input and output of Haskell Bidirectional Reference



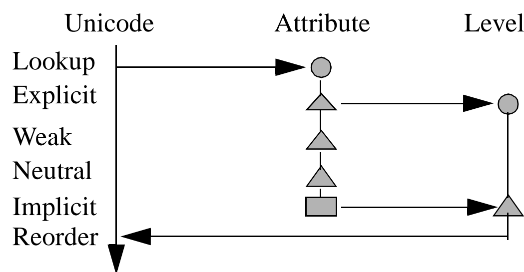
Each Unicode character has an associated Bidirectional attribute and level number. Figure 4-7 shows the general relationship of this information throughout the

---

1. The Haskell 98 Report defines the Char type as an enumeration consisting of 16-bit values conforming to the Unicode standard. The escape sequence used is consistent with that of Java (\u*hhhh*). Unicode is permitted in any identifier or any other place in a program. Currently the only Haskell implementation known to support Unicode directly is the Chalmers' Haskell Interpreter and Compiler.

steps of the algorithm. The first step in our implementation is to lookup and assign bidirectional attributes to the logical character stream. The attributes are obtained from the online character database as published in Unicode 3.0. At this point explicit processing assigns level numbers as well as honoring any directional overrides. Weak and neutral processing potentially causes attribute types to change based upon surrounding attribute types. Implicit processing assigns final level numbers to the streams which control reordering. Reordering then produces a sequence of Unicode characters in display order.

Figure 4-7. Data flow



HaBi uses the following three internal types:

- type Attributed = (Ucs4, Bidi)
- type Level= (Int, Ucs4, Bidi)
- data Run = LL[Level] | LR[Level] | RR[Level] | RL[Level]

Wherever possible the implementation treats characters collectively as sequential runs rather than as individual characters [1]. By using one of data type Run's four possible type constructors, characters can then be grouped by level. These four constructors signify the possible combinations of starting and ending run directions. For example, the LL constructor signifies that the start of a run and the end of a run are both left-to-right. Therefore runs of LL followed by RL are not created.

Before the details of the source code are discussed it is important to make note of the following concerning HaBi:

- The logical text stream is assumed to have already been separated into paragraphs and lines.
- Directional control codes are removed once processed.
- No limit is imposed on the number of allowable embeddings.
- Mirroring is accomplished by performing character replacement.

By separating those facets of layout dealing with reordering from those that are concerned with rendering (line breaking, glyph selection, and shaping) it becomes easier to understand the Haskell implementation.

4.6.1.1 HaBi Source Code

In the source code, functions are named in such a way so as to correspond to the appropriate section in the Unicode Bidirectional textual reference [100]. See Appendix A. For example, the function named weak refers to overall weak resolution. While the function named w1_7 lines 46-72 specifically refers to steps 1 through 7 in weak resolution.

The function logicalToDisplay lines 152-160 in Appendix A, is used to convert a stream in logical order to one in display order. First, calls to the functions explicit lines 37-41, weak lines 74-79, neutral lines 95-100, and implicit lines 115-120 form runs of fully resolved characters. Calls to reorder lines 135-141 and mirror lines 144-150 are then applied to the fully resolved runs which in turn yield a stream in display order. This is discussed in greater detail in the next few paragraphs.

The explicit function breaks the logical text stream into logical runs via calls to p2_3 lines1-8, x2_9 lines 10-27 and x10 lines 29-35. The reference description suggests the use of stacks for keeping track of levels, overrides, and embeddings. In our implementation stacks are used as well, but they are implicit rather than explicit (function x2_9 arguments two, three, and four). The functions weak, neutral, and implicit are then mapped onto each individual run.

In weak steps 1 though 7 lines 46-72 two pieces of information are carried forward (the second and third arguments of function w1_7) the current directional state and the last character's type. There are cases in the algorithm where a character's direction gets changed but the character's intrinsic type remains unchanged. For example, if a stream contained an AL followed by a EN the AL would change to type R (step three in weak types resolution). However the last character would need to remain AL so as to cause the EN to change to AN (step two in resolution of weak types).

The functions n1_2 lines 81-93 and i1_2 lines 103-113 resolve the neutral and implicit character types respectively. The details of these functions are not discussed as they are fairly straight forward. At this point runs are fully resolved and ready for reordering (function reorder).

Reordering occurs in two stages. In the first stage (function reverseRun lines 122-127), a run is either completely reversed or left as is. This decision is based upon whether a run's level is even or odd. If it is odd (right-to-left) then it is reversed. In the second stage (function reverseLevels lines 129-133), the list of runs are reordered. At first it may not be obvious that the list being folded is not the list of runs, but is the list of levels (highest level to the lowest odd level in the stream). Once reordering is finished the list of runs are collapsed into a single list of characters in display order.

4.6.1.2 Benefits of HaBi

By using a functional language we are able to separate details that are not directly related to the algorithm. In HaBi reordering is completely independent from character encoding. It does not matter what character encoding one uses (UCS4, UCS2, or UTF8). The Haskell type system and HaBi character attribute function allows the character encoding to change while not impacting the reordering algorithm.

Other implementations may find this level of separation difficult to achieve (Java and C). In C the size of types are not guaranteed to be portable, making C unsuitable as a reference. In the Java reference implementation the ramifications of moving to UCS4 are unclear. Our reference presents the steps as simple, easy to understand functions without side effects. This allows implementers to comprehend the true meaning of each step in the algorithm independently of the others while remaining free from language implementation details. The creation of test cases is thus more systematic.

4.7 Problems With Bidirectional Layout Algorithms

The biggest hindrance to the creation of a mechanism for converting logical data streams to display streams lies in the problem description. The problem of bidirectional layout is ill defined with respect to the input(s) and output(s).

Certainly the most obvious input is the data stream itself. But several situations require additional input in order to correctly determine the output stream. For example, in Farsi, mathematical expressions are written left-to-right while in Arabic they are written right-to-left [41]. This may require a special sub input (directional control code) to appear within the stream for proper handling to occur. If it becomes necessary to use control codes for obtaining the desired results, the purpose of an algorithm becomes unclear.

The problem of converting logical data streams to display streams is more confounding when one considers other possible inputs (paragraph levels, line breaks, shaping, directional overrides, numeric overrides, etc.) Are they to be treated as separate inputs? If they are treated as being distinct, when, where and how should they be used?

Determining the output(s) is not simple either. The correct output(s) is largely based on the context in which an algorithm will be used. If an algorithm is used to

render text, then appropriate outputs might be a glyph vector and a set of screen positions. On the other hand, if an algorithm is simply being used to determine character reordering, then an acceptable output might just be a reordered character stream.

4.7.1 Unicode Bidirectional Algorithm

The Unicode Bidirectional algorithm has gone through several iterations over the years. The current textual reference has been greatly refined. Nevertheless, we believe that there is still room for improvement. Implementing a bidirectional layout algorithm is not a trivial matter even when one restricts an implementation to just reordering. Part of the difficulty can be attributed to the textual description of the algorithm. Additionally there are areas that require further clarification.

As an example, consider step L2 of the Unicode Bidirectional Reference Algorithm. It states the following, “From the highest level found in the text to the lowest odd level on each line, reverse any contiguous sequence of characters that are at that level or higher. [100]” This has more than one possible interpretation. It could mean that once the highest level has been found and processed the next level for processing should be one less than the current level. It could also be interpreted as meaning that the next level to be processed is the next lowest level actually present in the text, which may be greater than one less than the current level. It was only through an examination of Unicode’s Java implementation that we were able to determine the answer. (The next level is one less than the current.)

There are also problems concerning the bounds of the Unicode Bidirectional Algorithm. In the absence of higher-order protocols it is not always possible to perform all the steps of the Unicode Bidirectional Algorithm. In particular, step L4 requires mirrored characters to be depicted by mirrored glyphs if their resolved directionality is R. However, glyph selection requires knowledge of fonts and glyph substitution tables. One possible mechanism for avoiding glyph substitutions is to perform mirroring via character substitutions. In this approach mirrored characters

are replaced by their corresponding character mirrors. In most situations this approach yields the same results. The only drawback occurs when a mirrored character does not have its corresponding mirror encoded in Unicode. For example, the square root character (U221A) does not have its corresponding mirror encoded.

When the Unicode Bidirectional Algorithm performs contextual analysis on text it overrides the static proprieties assigned to some of the characters. This occurs during the processing of weak and neutral types. Separating this portion of the algorithm from resolving implicit levels and reordering levels greatly extends the applicability of the algorithm. Ideally the analysis of the text should be distinct from the actual determination of directional boundaries.

Domain names, mathematical expressions, phone numbers, and other higher-order data elements are detected during the analysis phase. Nevertheless, it is impossible to create an algorithm that can always correctly identify such elements. The real issue is whether or not it is possible to create an algorithm that identifies such elements within some reasonable range of error and under a set of acceptable constraints for the elements themselves.

4.7.2 Reference Implementation

We argue that if source code is now going to serve as a reference we should pick source code that is more attuned to describing algorithms. We claim to have provided such a reference through the use of Haskell 98. Our HaBi reference is clear and succinct. The total number of lines of source code for the complete solution is less than 300 lines. The Unicode Java reference implementation is over 1000 lines [100].

4.7.3 HaBi

Even though, HaBi is a great improvement over current imperative implementations the functional approach has offered only limited success. Our original goal was to discover the true nature of bidirectional display, in hopes of producing a

more succinct algorithm. Unfortunately, we have a failed in this attempt. The algorithm appears to be ad hoc, hence it is not very algorithmic.

4.8 Limitations of Strategies

We believe, however, that the real problem of bidirectional layout lies not in the implementations, but rather in the goals of the algorithm. The requirements of a bidirectional layout algorithm are difficult to define. There are numerous interactions between higher-order protocols (line breaking, glyph selection, mirroring) and bidirectional layout. Additionally, there is a limited amount of complex bidirectional text from which to gather a strong consensus, making verification of results extremely difficult.

The major issues that all of the current algorithms suffer from are:

- Lack of separation between inferencing (script boundary detection) and reordering.
- Incorrect output — Reordering only makes sense in the context of glyphs.
- Inadequate input — there is simply not enough semantic information in plain text to properly determine directional boundaries.

4.8.1 Metadata

We believe that one of the best strategies to overcoming these limitations is to introduce character metadata into a plain text encoding. This enables additional semantic information to be expressed in a plain text stream as well as allowing for a clear separation between inferencing and reordering. This is the topic of discussion in the next chapter.

5 Enhancing Plain Text

In the previous chapter we concluded that the current Unicode based bidirectional layout algorithms: One, fail to separate inferencing from reordering and two, produce the wrong output. The problem is poor design and a lack of clear levels of abstraction. Code points are used for many different purposes. How should text for natural languages be represented? Is a stream of code points adequate? We think the answer is no. To overcome this deficiency, we examine strategies for enhancing plain text. These enhancements involve mechanisms for describing higher-order protocols. We generally refer to these enhancements as “metadata” (data describing data).

Algorithms that manipulate Unicode should be based upon on code points and character attributes, if possible, given that Unicode is a character encoding system. The Unicode Bidirectional Algorithm, among others, is a text algorithm that requires additional input and output (higher-order semantics) over and above the actual code points. Unicode wishes to define such algorithms, however it lacks a general mechanism for universally encoding higher-order semantics. Encoding higher-order semantics into Unicode would permit a cleaner division of responsibilities. Algorithms could be recast to take advantage of this division. To prove this is viable we recast the HaBi algorithm to take advantage of this division, separating the responsibility of determining directional boundaries (inferencing) from reordering.

Below we present the historical use of metadata within character encodings. This is followed by an examination of the presently available paradigms for expressing metadata. Particular attention is given to both Unicode’s character/control/metadata model and XML. We then present a universal framework for expressing

higher-order protocols within Unicode. Finally, the chapter concludes with evidence demonstrating the benefits and adaptability of the new approach.

5.1 Metadata

The need for expressing metadata has existed ever since humans started communicating with each other. Metadata is primarily expressed through our verbal speech. The tone, volume, and speed in which something is spoken often signals its importance or underlying emotion. Often this is more important than the data itself, and more difficult to codify.

Writing and printing systems also have their need for metadata. This metadata has been variously conveyed through the use of color, style, and size of glyphs. Initially metadata was used as a mechanism for circumventing the limitations of early encoding schemes. As our communication mechanisms advanced so did our need for expressing metadata.

5.1.1 Historical Perspective

One of the earliest uses of metadata appears in Baudot's 5-bit teleprinter. Baudot divided his character set into two distinct planes, named Letters and Figures. The Letters plane contained all the Uppercase Latin letters while the Figures plane contained the Arabic numerals and punctuation characters. Together, these two planes shared a single set of code values. To distinguish their meaning Baudot introduced two special meta-characters, letter shift "LTRS" and figure shift "FIGS". Whenever code points were transmitted they were preceded by either a FIGS or LTRS character. This enabled unambiguous interpretation of characters. This is similar to the shift lock mechanism in typewriters. For example, line 1 of Figure 5-1 spells out "BAUDOT" while line 2 spells out "?-7$95".[49],[18]

Figure 5-1. Using LTRS and FIGS in Baudot code

$$\text{0x1F 0x19 0x03 0x07 0x09 0x18 0x10} \quad \text{BAUDOT} \tag{1}$$

$$\text{0x1B 0x19 0x03 0x07 0x09 0x18 0x10} \quad \text{?-7\$95} \tag{2}$$

This still left the problem of how to transmit a special signal to a teleprinter operator. Baudot once again set aside a special code point, named bell “BEL”. This code point would not result in anything being printed, but rather it would be recognized by the physical teleprinter as the command to ring a bell. [49]

About 1900, we see code points being used as format effectors (code points that control the positioning of printed characters on a page). A good example of such usage can be seen in Murray’s code. Murray’s code introduced two additional characters column “COL”, and line page “LINE PAGE”. Known in International Telegraphy Alphabet Number 2 (ITA2) as carriage return and line feed. These characters were used to control the positioning of the rotating typehead and to control the advancement of paper. Murray’s encoding scheme was used for nearly fifty years with little modification. It also served as the foundation for future encoding techniques.[49]

During the late 1950s and early 1960s telecommunication hardware rapidly became complex. Consequently, hardware manufacturers needed more highly sophisticated protocols and greater amounts of metadata. For this purpose the US Army introduced a 6-bit character code called FIELDATA. FIELDATA was the first encoding to formally introduce the concept of supervisor codes, known today has control codes. These code points were used to signal communications hardware.[49]

The hardware manufacturers were certainly not alone in their need for metadata. The data processing community soon realized that they also had a need for metadata. This unfortunately taxed the existing encoding schemes (5-bit and 6-bit) so much so as to render them unusable. As a result, richer and more flexible encoding

schemes were created, the prime example being the American Standard Code for Information Interchange (ASCII). [12]

ASCII with its 7-bit encoding, served not only as a mechanism for data interchange, but also had many other special features. One feature was a mechanism for metadata. This metadata could be used for communicating higher-order protocols in both hardware and software. The architecture is based upon ASCII's escape character "ESC" at hex value 0x1B. Initially the ESC was used for shifting between character sets. This was of a particular importance to ALGOL programmers. For example, it allowed *if* to be used as both an identifier and a reserved word simultaneously. The presence of the ESC indicated that the *if* should be treated as a reserved word, rather than as an identifier [28]. As ASCII was adopted internationally the ESC became useful for signaling the swapping in and out of international character sets. This concept was expanded upon in 1980s in the ISO-2022 standard. [15],[54],[75]

ISO-2022 is an architecture and registration scheme for intermixing multiple 7-bit or 8-bit encodings using a modal encoding system similar to Baudot's. Escape sequences or special characters are used to switch between different character sets or between multiple versions of the same character set. This scheme operates in two phases. The first phase handles the switching between character sets, while the second handles the actual characters that make up the text.[57]

Non-modal encoding systems by contrast make direct use of the byte values in determining the size of a character. In non-modal encoding systems characters may vary in size within a stream of text; characters typically range from one to four bytes. This occurs in both the UTF-8 and UTF-16 encodings. [57]

In ISO-2022 up to four different sets of graphical characters may be simultaneously available, labeled G0 through G3. Escape sequences are used to assign and switch between the individual graphical sets. For example, line 1 of Figure 5-2 shows

the byte sequence for assigning the ASCII encoding to the G0 alternate graphic character set. Line 2 shows the Latin 1 encoding being assigned to the G1 set.[3],[75]

Figure 5-2. ISO-2022 escape sequences

ESC 0x28 0x42	assign ASCII to G0	**(1)**
ESC 0x2D 0x41	assign Latin 1 to G1	**(2)**

Most data processing tools make little if any distinction amongst data types. Data processing tools simply view information as bytes leaving the meaning of the data entirely open to human interpretation. For example, "UNIX grep" assumes that data is represented as a linear sequence of stateless fixed length independent bytes. Grep is highly flexible when it comes to searching characters or object code. This model has served text processing well under the assumption that one character equals one code point, but encoding systems have advanced and user expectations have risen.[76]

Over the last ten or so years Unicode has become the de facto standard for encoding multilingual text. This has brought a host of new outcomes that few could have imagined even a decade ago. Despite this advance, users want more than just enough information for intelligible communication. Plain text in its least common denominator is simply insufficient.

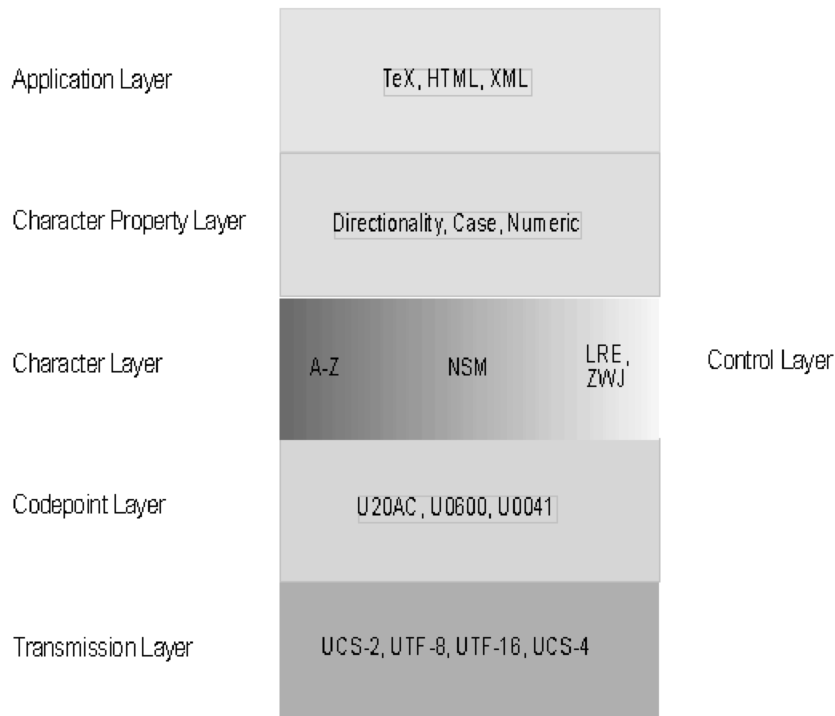
There have been several discussions and a few attempts to enrich plain text; ISO-2022 is one, XML can also be viewed in this framework. Both concern meta information yet have different purposes, goals, and audiences. The transition from storing and transmitting text as plain streams of code points is now well underway.[23]

5.2 Unicode Character Model

Figure 5-3 presents what might be called the Unicode character model. It (like the network model) has a transmission layer at the bottom and an application layer at

the top. The layers in between are the focus of this chapter. The Character/Control layer is intentionally depicted in a gradient fashion to illustrate the vague boundary separating characters from control. This lack of separation has made it more difficult to write applications that use Unicode.

Figure 5-3. Unicode Character Model



5.2.1 Transmission Layer

The transmission of text across text processes ranges from simple byte exchange to stateful encoding transmission. In chapter 3 we discussed in detail the various ways in which Unicode/ISO-10646 characters can be transmitted.

5.2.2 Code Point Layer

The most frequently occurring method of character identification is by numeric value. This approach has formed the hallmark of character encoding

schemes for nearly forty years. An example of which is the ASCII encoding scheme. Ideally the code point layer maps binary values to abstract character names.

In ASCII by contrast, the only unambiguous way to identify characters is by their numeric value. The use of abstract character names within ASCII has never been standardized, limiting our ability to unambiguously refer to characters. Consider the code point 0x5F in ASCII. This code point has been referred to as: “spacing underscore”, “low line”, and “horizontal bar”. This may not seem problematic as their glyphs are the same. This situation, however, has caused characters to be treated as if they were glyphs rather than as abstract entities, making the distinction between characters and glyphs almost impossible.

Additionally, having the highest level of abstraction limited to just the Code Point Layer does not permit a clean separation between text processes and the characters they manipulate. This problem is quite evident in parsing ASCII files. In particular consider the numerous ways in which a line end character may be expressed.

Alternatively characters can be identified by their abstract name or binary sequence. This technique is employed within ISO-10646. ISO-10646 provides a one-to-one mapping from code points to character names. This allows characters to be referred to unambiguously by either their abstract name or binary value. For example consider once again the code point value 0x5F. In ISO-10646 this code point value has but one name, “LOW LINE”.

This additional layer of abstraction still leaves open the question what does a character’s name mean. In ISO-10646 characters do not posses any properties other than their name. Unfortunately this places the burden of assigning properties into the hands of text processes, resulting in wide variation.

5.2.3 Character/Control Layer

Unicode defines the term character as "The smallest component of written language that has semantic value; refers to the abstract meaning and/or shape, rather than a specific shape." This definition is at odds with what is actually present within Unicode. There are several characters defined within Unicode that do not belong to any written language "controls" (characters that cause special actions to occur, but are not considered part of the data) as well as characters that specifically convey a definitive typographic shape "glyphs", Table 5-1. This intermixing has made it nearly impossible to determine when a character should be treated as data, metadata, or a glyph. Furthermore, the method by which new control codes are introduced is completely ad hoc. There is no standard range within Unicode that contains just controls. [96]

Table 5-1. Problem Characters

Type	Code Point	Purpose
OBJ REPLACE	0xFFFC	placeholder for external objects
ANNOTATION	0xFFF9 - 0xFFFB	used for formatting ruby characters in Japanese
CHAR REPLACE	0xFFFD	placeholder for non convertible characters
CONTROL	0x0000 - 0x001F, 0x007F - 0x0096	C0 and C1 control characters, used for legacy
NON BREAKING SPACES	0x00A0, 0xFEFF, 0x202F	non breaking space
NSM	0x0300 - 0x0362	non spacing diacritical marks used to form glyphs
SPACES	0x2000 - 0x200A, 0x200B, 0x3000	en, em, thin, hair, zero width, ideographic spaces
SEPARATORS	0x2028 - 0x2029	line and paragraph separator

Table 5-1. Problem Characters (Continued)

Type	Code Point	Purpose
ZWNJ	0x200C	zero width non joiner, prevent ligature formation
ZWJ	0x200D	zero width joiner, promote ligature formation
LRM	0x200E	left-to-right mark, used in bidi processing
RLM	0x200F	right-to-left mark, used in bidi processing
BIDI CONTROLS	0x202A - 0x202E	used for overriding directional properties, and for embedding directional runs
SHAPING	0x206A - 0x206F	used to control Arabic shaping
PRESENTATION FORMS	0xFB50 - 0xFDFF, 0xFE70 - 0xFEFE	Arabic ligatures and glyph variants used by rendering engines
HALFWIDTH AND FULLWIDTH	0xFF00 - 0xFFEF	half-width and full-width forms used in rendering engines

5.2.4 Character Property Layer

Clearly there exists a need for character properties. Text processes want to be able to interchange and interpret text unambiguously. Unicode adds an additional layer of abstraction onto ISO-10646. In Unicode each character may posses properties from three general areas, “normative”, “informative”, and “directional”. For example, see Table 5-2. [32]

Table 5-2. Character Properties

Code Point	Character Name	Normative	Informative	Directional
0x005F	LOW LINE		punctuation connector	neutral
0x0061	LATIN SMALL LETTER A	lowercase letter		left to right
0x0661	ARABIC-INDIC DIGIT ONE	decimal digit		arabic numeral

5.3 Strategies for Capturing Semantics

There are two general approaches to encoding higher-order semantics within text streams: *in-band signaling* and o*ut-of-band signaling*.

Using in-band signaling determining whether a character is data or metadata depends on the context in which a character is found. That is, code points are overloaded. This achieves maximal use of the character encoding, as characters are not duplicated. It also does not require encoding modifications as protocols change. All of this progress comes at the expense of parsing. It is no longer possible to conduct a simple parse of a stream looking for just data or metadata. This technique is employed within the HTML and XML protocols. [99]

Using out-of-band signaling for describing Unicode metadata requires the definition and transmission of complex structures similar to document data type definitions (DTD) in XML. This has the ill effect of making the transmission of Unicode more intricate. It would no longer be acceptable to simply transmit the raw Unicode text. Without the metadata, the meaning of the raw text may be ambiguous. On the other hand, parsing of data and metadata would be trivial, given that the two are not intermixed. [99]

5.3.1 XML

The extensible markup language (XML) provides a standard way of sharing structured documents and for defining other markup languages. XML uses Unicode as its character encoding for data and markup. Control codes, data characters, and markup characters may appear intermixed in a text stream. Confusion may ensue when control codes, data characters, and markup characters are combined with overlapping higher-order protocols. Additionally, there may be situations in which markup and control should not be interleaved. This issue is quickly being realized both within XML and Unicode. [23]

Whitespace characters in XML are used in both markup and data. The characters used in XML to represent whitespace are limited to “space”, “tab”, “carriage return”, and “line feed”. Unicode on the other hand, offers several characters for representing whitespace the (e.g., line separator U2028 and the paragraph separator U2029). The use of U2028 and U2029 within XML may lead to confusion because of the conflicting semantics. [33]

In Unicode these characters may be used to indicate hard line breaks and paragraphs within a data stream. These may affect visual rendering as well as acting as separators. When used within XML it is unclear whether the implied semantics can be ignored. Does the presence of one of these control codes indicate that a rendering protocol is being specified in addition to their use as whitespace, or are they simply whitespace? [23]

XML completely excludes certain Unicode characters from names (tags). The characters in the compatibility area and specials area UF900-UFFFE from Unicode are not permitted to be used within XML names. Their exclusion is due in part to the characters being already encoded in other places within Unicode. This is by no means the only reason. If characters from the compatibility area were included the issue of normalization would then to be addressed. (In this context normalization refers to name equivalence). [33]

Unicode provides guidelines and an algorithm for determining when two character sequences are equivalent. Unicode attempts to address this issue in Unicode Technical Report #15 Unicode Normalization Forms. In general there are two kinds of normalization, Canonical and Compatibility. [101]

Canonical normalization handles equivalence between decomposed and precomposed characters. This type of normalization is reversible. Compatibility normalization addresses equivalence between characters that do not visually appear the same. This type of normalization is irreversible.[32],[101]

Compatibility normalization in particular is problematic within XML. XML is designed to represent raw data free from any particular preferred presentation. Characters that may be compatible do not necessarily share the same semantics [32]. It may be the case that an additional protocol is being specified within the stream. For example, the UFB00 character on line 1 of Figure 5-4 is compatible with the two character sequence "U0066 U0066" on line 2 of Figure 5-4. Line 1 also specifies an additional protocol; in this case ligatures. In such a situation it is unclear whether or not the names were intended to be distinct. It is difficult to tell when the control function (higher-order protocol specification) of a character can be ignored and when it can't. Additionally, some have argued that Unicode's Normalization Algorithm is difficult to implement, resource intensive, and prone to errors. To avoid such problems XML has chosen not to perform normalization when comparing names. [18],[33]

Figure 5-4. Compatibility Normalization

UFB00	ff ligature	**(1)**
U0066 U0066	ff no ligature	**(2)**

Problems such as these are due to the lack of separation between syntax and semantics within Unicode. The absence of a general mechanism for specifying higher-order protocols "metadata" only serves to further confound these issues.[18],[33]

5.3.2 Language Tagging

Over the years there has become a need for language "tagging" of multilingual text. In some cases such information is necessary for correct behavior. For example, language information becomes necessary when an intermixed stream of Japanese, Chinese, and Korean is to be rendered. In this case, the unified Han char-

acters need to be displayed using language specific glyph variants. Without higher-order language information it becomes difficult to select the appropriate glyphs.

Recently Unicode has added a mechanism for encoding language information within plain text. This is achieved in Unicode through the introduction of a special set of characters that may be used for language tagging. The current strategy under consideration within Unicode is to add 97 new characters to Unicode. These characters would be comprised of copies of the ASCII graphic characters, a language character tag, and a cancel tag character. These characters would be encoded in Plane 14 “surrogates” U000E0000 - U000E007F.[99]

The use of the tags in Unicode is very simple. First a tag identifier character is chosen, followed by an arbitrary number of unicode tag characters. A tag is implicitly terminated when either a non tag character is found or another tag identifier is encountered. Currently there is only one tag identifier defined, the language tag. See Figure 5-5. Line 1 of Figure 5-5 demonstrates the use of the fixed code point language tag “U000E0001” along with the cancel tag “U000E007F”. The plane 14 ASCII graphic characters are in bold and are used to identify the language. The language name is formed by concatenating the language id from ISO-639 and the country code from ISO-3166. In the future a generic tag identifier may be added for private tag definitions. [99]

Figure 5-5. Language tag

U000E0001 **fr-Fr** french text U000E0001 U000E007F **(1)**

Tag values can be cancelled by using the tag cancel character. The cancel character is simply appended onto a tag identifier. This has the effect of cancelling that tag identifier’s value. If the cancel tag is transmitted without a tag identifier the effect is to cancel any and all processed tag values. [99]

The value of a tag continues until either it implicitly goes out of scope or a cancel tag character is found. Tags of the same type may not be nested. The occurrence of two consecutive tag types simply applies the new value to the rest of the unprocessed stream. Tags of differing types may be interlocked. Tags of different types are assumed to ignore each other; there are no dependencies between tags. [99]

Tag characters have no particular visible rendering and have no direct affect on the layout of a stream. Tag aware processes may chose to format streams according to their own interpretation of tags and their associated values. Tag unaware processes should leave tag data alone and continue processing. [99]

5.3.2.1 Directional Properties of Language Tags

In this section we show that language tags can interact with other facets of Unicode including the Bidirectional Algorithm.

In Unicode, language tag characters are all marked as having a neutral direction. Neutral characters pick up their direction from the surrounding strong characters (left or right). This seems reasonable as we do not wish the tags to accidentally influence the Bidirectional Algorithm. If the direction of the tags were left-to-right or right-to-left, rather than neutral, then the tags would influence the resolution of weak and neutral types due to their juxtaposition. [96]

The example in Figure 5-6 demonstrates this error. In Figure 5-6 Arabic characters are represented in upper case. The character sequence **LANGar** (Arabic language tag) in bold is a visual representation for the Unicode sequence (UE0001,UE0061,UE0072). In this example assume that the language tag characters are assigned the left-to-right direction. Line 1 is a sequence of characters in logical order, while line 2 is the expected resultant display ordering. The display ordering in line 3 is incorrect, because the tag characters inadvertently participated in bidirectional processing. Marking language tags as neutral makes sense in this framework of protecting the Bidirectional Algorithm.

Ironically, in the process of protecting the Bidirectional Algorithm we inadvertently allowed the Bidirectional Algorithm to influence language tags. For example, line 1 on Figure 5-7 is a sequence of Arabic characters in logical order with an embedded language tag, Urdu (UE0001,UE0075,UE0072). Line 2 is the same sequence of characters, but in display order (output from the Bidirectional Algorithm). When the character sequence is rendered the language tag, however, is displayed backwards and appears as "ru" which indicates Russian. The reason the language tag is displayed backwards is, because neutral characters pick up their direction form the surrounding characters, in this case right-to-left. Clearly this is undesirable and therefore must be prevented.

Figure 5-6. Error in bidirectional processing

CIBARA **LANGar**, 123 **(1)**

123 , **LANGar** ARABIC **(2)**

LANGar , 123 ARABIC **(3)**

Figure 5-7. Error in language tagging

U0624,UE0001,UE0075,UE0072,U00623 — lang ur (Urdu), logical **(1)**

U0623,UE0072,UE0075,UE0001,U0624 — lang ru (Russian), display **(2)**

This problem can be solved by introducing a new bidirectional property, "ignore". This will enable the Bidirectional Algorithm to continue to function properly while also protecting the semantics of tags. Characters that posses the direction ignore will not have any direction and will not pick up any surrounding direction, preventing these characters from participating in the Bidirectional Algorithm. [96],[100]

5.3.3 General Unicode Metadata

It is still possible to construct a metadata signaling mechanism for the specific purpose of mixing data and metadata and yet allow for simple parsing. This is called

“light-weight in-band signalling”. This is the approach that Unicode has adopted for language tagging. [99]

The light-weight approach is useful, but Unicode’s application of it creates two problems. First, new tag identifiers always require the introduction of a new Unicode code point. This puts Unicode in a constant state of flux as well as fixing the number of possible tag identifiers. Second, there is no way to specify multiple parameters for a tag. This deficiency forces the creation of additional tag identifiers to circumvent this limitation.

We propose that a more generalized approach be taken. Our design philosophy is to encode a minimal set of stateless metadata characters that enable the definition of higher-order protocols. Our use of the term stateless in this context refers to whether a character’s type (data or metadata) can be determined by its code point value alone. In some metadata systems (HTML and XML) a character’s type can only be determined by examining the full context in which it appears. Unfortunately, these systems require sophisticated parsing techniques. We argue for the use of stateless characters in our system because they simplify both parsing and understanding.

Naturally, the construction of a metadata encoding system requires a balancing of trade-offs. One such trade-off is whether or not to define a syntax for the definition of tags. We believe that we should specify a minimal and flexible syntax for tags that allows for unambiguous communication, yet does not impose any particular style of protocol such as HTML.

In choosing the set of metadata characters we suggest that we keep the copy of the ASCII graphic characters that are used in Unicode’s language tagging model. We should however remove the fixed code point tag identifiers. In their place we introduce two new characters, tag delimiter U000E0001 and tag argument separator U000E0002. See Table 5-3. The motivation for these characters comes from the SGML/XML/HTML camps. These characters provide an easy migration path for

embedding XML like protocols within Unicode. The use of these characters is by no means required—applications may chose alternative methods. On the other hand, the use of the tag delimiter and the tag argument separator help prevent confusion between whether a sequence of tag characters represents a tag name or tag argument. Additionally, the use of these characters guarantees that tag neutral tools can be created. Such tools can always count on the fact that consecutive tags are delimited and that their arguments are separated by tag argument separators.

Table 5-3. Tag characters

Tag Characters	**UCS-4**	**Visual Representation**	**Purpose**
delimiter	U000E0001	|	control
argument separator	U000E0002	@	control
space	U000E0020		display
graphic characters	U000E0021 - U000E007E	**a-z**, **A-Z**, **0-9**, etc.	display

The tag delimiter character is used to separate consecutive tags from one another. While the tag argument separator is used to delineate multiple tag arguments. This approach allows the same set of tag graphic characters to be used for both tag names and tag arguments. Additionally, tag names are spelled out rather than being assigned to a fixed single code point.

The overall construction of tags will still remain simple. First, the tag name is spelled out using the tag graphic characters, followed by an optional tag argument separator. Second, there may be an arbitrary number of tag arguments for each tag, each argument being separated by a tag argument separator. A tag name is terminated by either encountering a tag argument separator, or an optional tag delimiter. This still allows for relatively simple parsing. The regular expressions for tags, tag names, and tag arguments can be found on lines 4-7 in Figure 5-8. The usage of tags in text

streams is also very simple to comprehend. See the regular expression on line 2 in Figure 5-9.

Figure 5-8. Regular expressions for tags

<tag graphic character> ::= [U000E0020-U000E007E]	**(1)**
<tag delimiter> ::= U000E0001	**(2)**
<tag argument separator> ::= U000E0002	**(3)**
<tag name> ::= <tag graphic character>$^{+}$	**(4)**
<tag argument> ::= <tag argument separator><tag graphic character>$^{+}$	**(5)**
<tag> ::= <tag name><tag argument>*	**(6)**
<tag> ::= <tag delimiter>	**(7)**

Figure 5-9. Regular expression for text stream

<data character> ::= [U00000000-U0002FA1D]	**(1)**
<text stream> ::= (<data character>|<tag>)$^{+}$	**(2)**

Throughout this chapter tag characters are represented in boldface. Additionally the vertical bar "**|**" is used to depict the tag delimiter and the at sign "**@**" denotes the tag argument separator. See Table 5-3. For example, line 1 of Figure 5-10 shows a stream with two tags, "**XX**" and "**YY**". Additionally, the tag "**XX**" has one argument "**a**", and the "**YY**" tag has two arguments "**b**" and "**c**". The example suggests the nesting of "**YY**" within "**XX**". In this sample protocol we terminate the scope of a tag by repeating the tag name preceded by the tag graphic character solidus "**/**". This method of terminating scope is not a requirement, protocols may adopt other methods or none at all.

The semantics of such combinations are left to protocol designers rather than the metadata. This affords the greatest flexibility and yet still retains the ability to per-

form simple parsing. This design allows Unicode to simply be in the business of defining mechanism rather than mechanism and policy.

Figure 5-10. Sample tag

def**XX@a|YY@b@c**|ghi/**YY**|jkl/**XX**| **(1)**

It is foreseeable that Unicode would remain the registrar of tag identifiers, while working in conjunction with other standards bodies. Though, this does not preclude private tags from being defined for those cases in which widespread protocol adoption is not required.

Similarly, the semantics of cancelling or ending the scope of tags will also be left to the protocol designer. It is possible that in some protocols tag cancellation might undo the last tag, while in others it may end the scope of a tag. Additionally, there is no requirement that either of these interpretations be used at all.

The example in Figure 5-11 shows how the language tag would be represented in the new tagging model. Line 1 of Figure 5-11 is copied from Figure 5-5. Line 2 shows the language tag spelled out with the two tag arguments being clearly delineated. We suggest that the spelling out of tag names is a small price to pay for this enhanced functionality.

Figure 5-11. Alternative language tag

U000E0001 **fr-FR** french text U000E0001 U000E007F **(1)**

LANG@fr@FR| french text **/LANG|** **(2)**

5.4 Encoding XML

In this section we demonstrate through the use of examples how our metadata encoding system can be used to encode XML and XML like protocols. The primary syntactic elements, for which we provide mappings in our examples, include: tags,

entity references, and comments. We believe that these elements are sufficient for demonstrating that the mapping of XML to metadata is both feasible and simple.

The example in Figure 5-12 shows a sample address book modeled in XML. The address book contains a collection of contacts, in this case just one contact. Each contact in turn contains a name and address. In this example we see three general types of tags:

- Self closing tags — the *<business/>* tag on line 8 on Figure 5-12.
- Start tags — the *<addressbook>* tag on line 3 on Figure 5-12.
- End tags — the *</contact>* tag on line 13 on Figure 5-12.

Additionally, each tag may optionally contain arguments. For example, the *<contact type=”business”>* tag on line 5 on Figure 5-12, contains a single tag argument. In this tag the string *type* represents the name of the argument while the string *business* is its corresponding value. [72]

Figure 5-12. Sample XML code

```
1    <?xml version=”1.0”?>
2    <!ENTITY amp “&#38;”>
3    <addressbook>
4    <!- this is an address ->
5        <contact type=”business”>
6        <name>Steve&apos;s Bar &amp; Grill</name>
7        <nickname>&gt;Steve&lt;</nickname>
8        <business/>
9        <address>150 W. University Blvd</address>
10       <city>Melbourne</city>
11       <state>FL</state>
12       <zip>32901</zip>
13       </contact>
14   </addressbook>
```

The sample XML file also contains entity references. In XML entity references assign aliases to pieces of data. It serves as a unique reference to some XML data. Entity references are made by using an ampersand and a semicolon. For example, line 6 on Figure 5-12 shows a reference to the *amp* entity (ampersand), while the

definition of the reference is shown on line 2. Normally, these characters would be processed differently in XML. However, with entity references an XML parser does not get confused. [72]

In this example we also see the use of a comment. See line 4 on Figure 5-12. Comments are used to provide additional information about an XML document, however they are not actually part of the data in a XML document. Comments begin with a <! and end with ->. The only restriction XML places upon comments is that they do not contain double hyphens --, as they conflict with XML's comment syntax. [72]

In general when we map a XML document to combined data/metadata most characters remain unchanged. This is particularly true of characters that are not part of XML tags. Nevertheless, characters that make up an XML tag get mapped to a corresponding metadata graphic tag character. However, we believe that their is some flexibility in this mapping. In particular, the < and > do not need to be mapped at all. These characters are not needed, because a XML parser can now immediately tell whether a character is part of a tag or not simply by examining its code point value. Therefore, it is unnecessary to map these characters.

The text in Figure 5-13 is the same XML document from Figure 5-13 re-encoded using our metadata tagging system. In certain cases we have changed the syntax of the XML tags slightly to take advantage of our metadata system and to be more consistent with our general tag syntax. In particular, these changes can be seen on lines 1, 2 and 5. Additionally, we have created a tag specifically for the purpose of indicating a comment. See line 4 on Figure 5-13. To indicate whether a tag is a start tag, end tag, or a self closing tag we have adopted the following convention:

- If the tag is an end tag then the tag name is preceded by a solidus "/".
- If the tag is a self closing then tag the tag name is followed by a solidus.
- All remaining tags are assumed to be start tags.

It is important to note that this convention is not the only way in which the tag type can be expressed. It is quite possible that some other convention could be adopted. One alternative, would be to indicate the type of a tag as a tag argument, rather than as part of the name. We have chosen to indicate the type as part of the tag name, as this closely matches XML's syntax.

Figure 5-13. Sample XML code encoded in metadata

```
1   |?xml@version="1.0"|
2   |ENTITY@amp@CP@U@0026|
3   |addressbook|
4   |cmt@this is an address|
5       |contact@type="business"|
6       |name|Steve|&apos|s Bar|&amp| Grill|/name|
7       |nickname||&gt|Steve|&lt||/nickname|
8       |business/|
9       |address|150 W. University Blvd|/address|
10      |city|Melbourne|/city|
11      |state|FL|/state|
12      |zip|32901|/zip|
13      |/contact|
14  |/addressbook|
```

The remaining issue involving the mapping of XML to metadata deals with the representation of characters outside the graphic tag character range. In XML it is legal to use characters outside the ASCII range in tags and in data. This is possible, because XML uses Unicode as its native encoding system. It is possible to represent these characters in our metadata system using our code point protocol. See line 2 on Figure 5-13. The raw code point protocol embeds any Unicode code point in metadata. The **CP** tag identifies the code point protocol, **U** indicates the Unicode character set, and **0026** is the hexadecimal value of the Unicode code point.

We have chosen not to encode a special metadata escape character for the purpose of embedding characters, because the encoding of such a character violates one of the key goals in our metadata system, stateless encoding. In our metadata system the meaning of each character is unambiguous and independent from any other char-

acter. If we were to encode an escape character then the digits following it would need to be overloaded. Sometimes the digits would be tag digits and sometimes they would be escaped digits. This makes the meaning of the digits contextual. This kind of overloading and contextual behavior returns us to XML's problems, which our strategy has been engineered to avoid.

5.5 New XML

Encoding XML in our metadata system offers several advantages. First, the use of the tag delimiters "|" around each tag are actually unnecessary. The tag delimiter is only required for consecutive tags. Additionally, only one of these tag delimiters are necessary. Each tag is immediately detectable by simply examining its code point value. This is not possible in XML, hence they need to clearly indicate the bounds of each tag.

Second, in XML there are certain characters that need to be escaped so that they can be used in data. These characters include the less-than "<" and greater-than ">" characters. Normally, these characters are used for indicating the bounds of a tag. When they are meant to be interpreted as data and not as tag indicators, they must be referred to via entity references, so as to avoid confusing the XML parser. For example, the ">" and "<" entity references on line 7 on Figure 5-12. In this example the ">" refers to the greater-than character and the "<" refers to the less-than character. In our metadata scheme this is completely unnecessary. There is never any confusion as to whether a character represents data or metadata. Therefore, we do not need this mechanism at all in our form of XML.

5.6 Text Element

Now that a definition of a general metadata mechanism has been established tags other than language may be constructed. In the next section we see one such

example, the text element tag. This tag will enable the embedding of additional linguistic information into pain text streams.

Traditionally text processes manipulated ASCII data with the implicit understanding that every code point equated to a single character and in turn a single text element, which then served as a fundamental unit of manipulation. In most cases this assumption held, especially given that only English text was being processed. [96]

Multilingual information processing, however breaks the assumption that code points, characters, and text elements are all equal. Text elements are directly tied to a text process, script, and language. Encodings today provide an abstract set of characters directly mapped onto set of numerals. The abstract characters are then grouped to form text elements.[96]

In some cases a text element may still equate to a single character while in other situations a text element may be comprised of several characters. For example, in Spanish the character sequence “ll” is treated as a single text element when sorted, but is treated as two text elements “l” and “l” when printed. [96]

Unicode relies on an abstract notion of characters and text elements. Unfortunately, a general mechanism for indicating text elements is lacking. In some instances a text element is implicitly specified through a sequence of characters. For example, line 1 of Figure 5-14 shows how a base character and a non spacing diacritic combine to form a single text element. See line 2 of Figure 5-14.

Figure 5-14. Combining characters

U00D6	O¨	decomposed	**(1)**
U004F U0308	Ö	precomposed	**(2)**

In other cases text elements are explicitly specified by control codes. In particular, Unicode uses control codes (e.g., the zero width joiner U200D and the zero width non joiner U200C) for forming visual text elements. These characters affect

ligature formation and cursive connection of glyphs. The intended semantic of the zero width non joiner is to break cursive connections and ligatures. The zero width joiner is designed to form a more highly connected rendering of adjacent characters.[97]

For example, line 1 of Figure 5-15 shows the sequence of code points for constructing a ligature. The characters x and y represent arbitrary characters. Line 2 shows how the zero width non joiner can be used to break a cursive connection. Problems still arise when one wishes to suppress ligatures while still promoting cursive connections. In this situation Unicode recommends combining the zero width non joiner and the zero width joiner. See line 3 of Figure 5-15.[97]

Rather than using control codes with complicated semantics and implicit sequences of characters to form text elements, a simple generalized mechanism can be used. Nonetheless, Unicode has no general way to indicate that sequences of characters should be viewed as a single text element. The current approach relies on a higher-order protocol outside of Unicode, such as XML which is designed to describe the structure of documents and collections of data, not individual characters and text elements. XML requires data to strictly adhere to a hierarchical organization. This may be appropriate for documents, but can be troublesome for a simple text stream. The model that is really required needs to be organized around characters and text elements.

This can be achieved through metadata tags and simple protocols. For example, the zero width joiner and zero width non joiner characters can be described by a new tag; the text element “ELM”. The ELM tag is used to group multiple characters together so that they can be treated as a single grapheme or text element. For example, line 1 of Figure 5-16 shows a text element “xy” for all purposes.

When characters are grouped together it may be for the purpose of rendering, sorting, or case conversion. The purpose of the grouping does not need to be

understood by Unicode. The semantics should only be determined by processes that make direct use of such information. The tag is simply a conduit for signaling higher-order semantics.

For example, line 2 of Figure 5-16 shows a text element “xy” for the purposes of forming ligatures, but not searching or sorting. Line 3 demonstrates the text element “xy” being cursively connected, while suppressing ligature formation.

Additionally the ELM tag can be used to form other semantic groupings. For example, in Spanish when “c” is followed by “h” the two single characters combine to form the single text element “ch”. See line 4 of Figure 5-16. This grouping does not effect rendering, but has implications in sorting. In German however, groupings affect case conversion. For example, the character sequence “SS” when converted to lowercase results in the single character “ß”. See line 5 of Figure 5-16.[96]

Figure 5-15. Joiners

x U200D y **(1)**

x U200C y **(2)**

x U200D U200C U200D y **(3)**

Figure 5-16. ELM tag

ELM|xy/**ELM**| **(1)**

ELM@LIG|xy/**ELM**| **(2)**

ELM@JOIN|xy/**ELM**| **(3)**

ELM@COLL|ch/**ELM**| **(4)**

ELM@CASE|SS/**ELM**| **(5)**

Table 5-4 lists a few other types of tags that are also based upon the general text element tag. Each entry in Table 5-4 specifies a specific Unicode semantic construct and its associated metadata tag.

Table 5-4. Other text element tags

Semantic Construct	Metadata Tag
Ligatures	**ELM@LIG**|xy/**ELM**|
Glyph Variant	**ELM@FNT**|xy/**ELM**|
Non Breaking	**ELM@NBR**|xy/**ELM**|
Initial Form	**ELM@INI**|xy/**ELM**|
Medial Form	**ELM@MED**|xy/**ELM**|
Final Form	**ELM@FIN**|xy/**ELM**|
Isolated Form	**ELM@ISO**|xy/**ELM**|
Circle	**ELM@CIR**|xy/**ELM**|
Superscript	**ELM@SUP**|xy/**ELM**|
Subscript	**ELM@SUB**|xy/**ELM**|
Vertical	**ELM@VER**|xy/**ELM**|
Wide	**ELM@WID**|xy/**ELM**|
Narrow	**ELM@NAR**|xy/**ELM**|
Small	**ELM@SMA**|xy/**ELM**|
Square	**ELM@SQU**|xy/**ELM**|
Fraction	**ELM@FRA**|xy/**ELM**|

5.7 Metadata and Bidirectional Inferencing

Plain text streams that contain characters of varying direction pose a particular problem for determining the correct visual presentation. There are several instances in which it is nearly impossible to render bidirectional text correctly in the absence of any higher-order information. In particular, picking glyphs requires that a rendering engine have knowledge of fonts.

The Unicode Bidirectional Algorithm operates as a stream to stream conversion which is logical given that Unicode is a character encoding mechanism and not a glyph encoding scheme [96]. This output, however is insufficient by itself to

correctly display bidirectional text. If a process is going to present bidirectional text then the output must be glyphs and glyph positions. The Unicode Bidirectional algorithm can not possibly produce this output and, still remain consistent with Unicode's primary character encoding scheme.

The core of the Unicode Bidirectional algorithm is centered around three aspects: resolving character types, reordering characters and analyzing mirrors. The bidirectional algorithm is applied to each paragraph on a line by line basis. During resolution characters that do not have a strong direction are assigned a direction based on the surrounding characters or directional overrides. In the reordering phase sequences of characters are reversed as necessary to obtain the correct visual ordering. Finally, each mirrored character (parenthesis, brackets, braces, etc.) is examined to see if it needs to be replaced with its symmetric mirror.[96]

This algorithm causes an irreversible change to the input stream which is a significant flaw. The logical ordering is no longer available. This inhibits the construction of an algorithm that takes as input a stream in display order and produces as output its corresponding logical ordering. The example in Figure 5-17 illustrates this problem. In Figure 5-17 Arabic letters are depicted by upper case latin letters while the right square bracket indicates a right-to-left override (U202E). Line 1 is a stream in display order, lines 2 and 3 are streams in logical order. In either case if the Bidirectional Algorithm is applied to line 2 or line 3 the result is line 1.

Figure 5-17. Mapping from display order to logical order

123 (DCBA) **(1)**

(ABCD) 123 **(2)**

]123 (ABCD) **(3)**

It is also impossible to tell whether a stream has been processed by the Bidirectional Algorithm. The output does not contain any identifying markers to indicate

that a stream has been processed. A text process can never be sure whether an input stream has undergone bidirectional processing. To further complicate the situation the Bidirectional Algorithm must be applied on a line by line basis. This is not always easy to accomplish if display and font metrics are not available.

We introduce three tags for bidirectional processing: "PAR" paragraph, direction "DIR", and mirror "MIR". The PAR tag signifies the beginning of a paragraph. It takes one argument, the base direction of the paragraph either right "R" or left "L".

The DIR tag takes one argument as well, the resolved segment's direction either "L" or "R". The MIR tag does not require any argument. Its presence indicates that the preceding character should be replaced by its symmetric mirror. The scope of the DIR tag is terminated by either a PAR tag or the end of the input stream.

For example, in Figure 5-18 line 1 represents a stream of characters in logical order. Line 2 is the output stream after running the Bidirectional Algorithm using tagging. Arabic letters are represented by upper case Latin letters. Tag characters are indicated in bold. The at sign represents the tag argument separator and the vertical bar represents the tag separator "U000E0001". The output of the algorithm only inserts tags to indicate resolved directional boundaries and mirrors. The data characters still remain in logical order.

Furthermore, the bidirectional embedding controls "LRE", "RLE", "LRO", "RLO", and "PDF" can be eliminated because they are superseded by the DIR tag. These controls act solely as format effectors. They convey no other semantic information and are unnecessary when viewed in light of the DIR tag.

Figure 5-18. Example output stream

(ABCD)123 **(1)**

PAR@R|MIR|(ABCD**MIR|**)**DIR@L|**123**/DIR|/PAR/** **(2)**

The Bidirectional Algorithm only requires two changes to accommodate the new tags. In those places where the text is to be reversed a DIR tag is inserted to indicate the resultant direction rather than actually reversing the stream itself. In those places where a symmetric mirror is required a MIR tag is inserted to indicate that this character should be replaced with its corresponding mirror. The Haskell functions `tagLevel` and `tagRun` replace functions `reverseRun`, `reverseLevels` and `reorder`. See Appendix B lines 1-39. The `mirror` function has been changed to insert a MIR tag rather than directly replacing a character with its symmetric mirror.

The Bidirectional Algorithm could also be extended to directly interpret tags itself. This would be extremely beneficial in cases where the data and the implicit rules do not provide adequate results. For example, in Farsi mathematical expressions are written left-to-right while in Arabic they are written right-to-left.

Under the traditional Bidirectional Algorithm control codes would need to be inserted into the stream to force correct rendering. See line 1 Figure 5-19 where the characters “LRE” and “PDF” represent the Unicode control codes Left to Right Embedding and Pop Directional Format respectively [100].

The extended Bidirectional Algorithm would address this through the addition of two tags “MATH” and “LANG”. These tags would be inserted into the stream to identify the language and that portion of the stream that is a mathematical expression. By using tagging the output stream still remains in logical order with its direction correctly resolved without the need of control codes. See lines 2 and 3 of Figure 5-19.

Figure 5-19. Mathematical expression

LRE 1 + 1 = 2 PDF **(1)**

LANG@fa@IR|MATH| 1 + 1 = 2 **/MATH|** **(2)**

LANG@fa@IR|MATH|DIR@L| 1 + 1 = 2 **/MATH|/DIR|** **(3)**

5.7.0.1 HTML and Bidirectional Tags

The HTML 4.0 specification introduces a bidirectional override tag “BDO” for explicitly controlling the direction by which a tag’s contents should be displayed. Lines 1 and 2 of Figure 5-20 illustrate the syntax of this tag. This tag is currently supported in Microsoft’s Internet Explorer. [23]

Figure 5-20. BDO tag syntax

<bdo dir=”LTR”>body content</bdo> **(1)**

<bdo dir=”RTL”>body content</bdo> **(2)**

These tags can be used in conjunction with the Unicode bidirectional tags. The Unicode tags can be directly converted into the HTML bidirectional tags [23]. This allows for a clean division of responsibilities for displaying bidirectional data. The Unicode metadata tags simply serve as bidirectional markers. Browsers can then directly render the resultant HTML. This permits the Unicode bidirectional algorithm to be free from the problems of determining font and display metrics.

The UniMeta program takes as input a file encoded in UTF-8 which contains Unicode text in logical order with bidirectional tags. See Appendix C lines 1-99. The UniMeta program then converts the input text into HTML. Each unicode metadata tag is replaced with a corresponding HTML tag. Currently there is no corresponding tag for mirroring in HTML. When a Unicode MIR tag is found it is simply ignored. The example in Figure 5-21 illustrates the output from the UniMeta Java program. Lines 1 and 2 are copied from Figure 5-18. Line 3 is the resultant HTML with BDO tags.

Figure 5-21. Using HTML bidirectional tags

(ABCD)123 **(1)**

PAR@R|MIR|(ABCD**MIR|**)**DIR@L|**123**/DIR|/PAR|** **(2)**

<bdo dir=”rtl”>(ABCD) <bdo dir=”ltr”>123</bdo></bdo> **(3)**

By using metadata tags to implement the Bidirectional Algorithm a clear division of responsibilities is achieved. The bidirectional layout process is now divided into two separate and distinct phases, logical run determination “inferencing” and physical presentation “reordering”. This enables character data to remain in logical order, and still contain the necessary information for it to be correctly displayed. Additionally, any text process receiving such a stream is able to immediately detect that the stream has been bidirectionally processed.

5.8 New Architecture

The introduction of metadata into an encoding allows for a general reorganization of character coding systems. We refer to this reorganization as Metacode. In the next chapter we explore this new architecture in depth.

6 Metacode

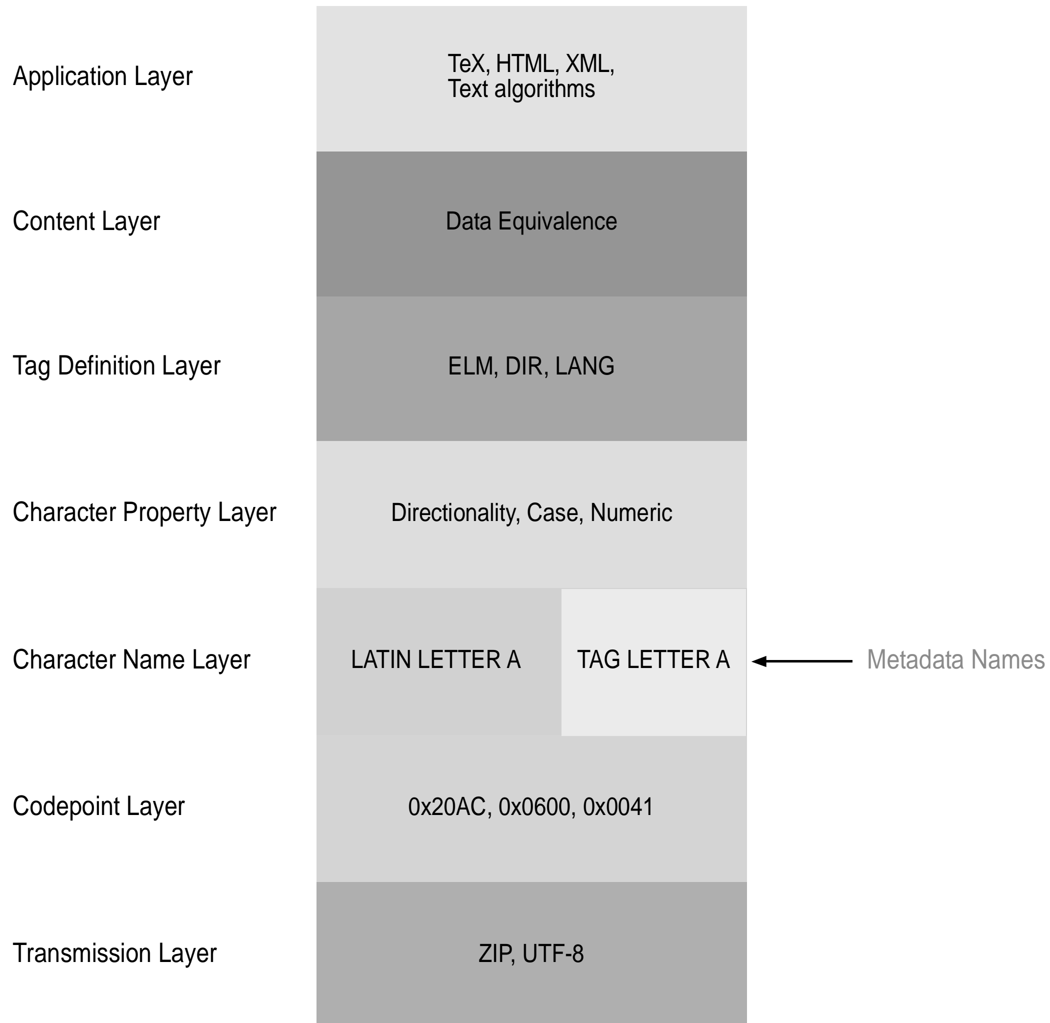
In this chapter we present both a new coded character set and text encoding framework that enables a separation of concerns, we call this the Metacode system. In the Metacode information processing system, concepts (policies) are separate and distinct from implementation (mechanism). Metacode at its core is an architecture for describing written natural language data. Metacode permits various ideas, concepts and policies to coexist, while still remaining efficient. The key advantage this new architecture offers over current models is its ability to unambiguously separate content from control. In Metacode only characters that express raw content are encoded. In particular, Metacode does not encode controls, ligatures, glyph variants, and half-width forms. By only encoding "pure" characters, Metacode places a greater emphasis on content. In this chapter we make recommendations and take the first steps towards implementing an architecture for multilingual information processing.

6.1 Metacode Architecture

Figure 6-1 presents the layers in the new text framework we propose. Metacode is built upon the same general principles used in the Open Systems Interconnection (OSI) network layer model [90]. In particular the architecture is designed around the following notions:

- A layer is created when a different level of abstraction is required.
- Each layer performs a well defined function.
- The number of layers is large enough to allow for a clean separation of responsibilities, but not so small as to group unrelated functions together out of necessity.

Figure 6-1. New Text Framework



Unlike other multilingual encoding systems which switch between various coded character sets, Metacode uses only one character set. In the Metacode system there are no special modes, states, or escape sequences. Metacode is strictly a fixed width character encoding scheme where each character is represented using the same number of bytes. Just as other multilingual text encoding systems, Metacode includes coverage for both the Asian and European writing systems.

Characters in Metacode are grouped into two broad categories, data and meta-data. We believe that most of Metacode's data characters would be comprised from

elements in natural written languages. In Metacode metadata characters provide a mechanism for describing higher-order information about data characters.

For nearly forty years, the most frequently occurring method of character identification has been by numeric value. This approach has formed the hallmark of character encoding schemes such as ASCII. Metacode continues this tradition.

In Metacode characters are identified by code point value. In Metacode code points are based upon an integer index. This index is used to map Metacode character names to binary sequences. This index should be large enough to accommodate the linguistic elements of both modern and ancient writing systems. Most researchers within the I18N community believe the total number of written characters will not exceed 2^{32}. It seems logical that 32-bits is an appropriate size for an index.

In Metacode, code points could be transmitted in two general ways. First, some form of binary encoding could be used, for example UTF-8. Second, a non-binary form of transmission could be adopted, for example character name transmission. That said, we anticipate most applications will use some form of binary transmission. Large multilingual encodings by their nature require a transmission layer. The ASCII character encoding scheme never needed a transmission layer, because encoding and transmission were synonymous. Nevertheless, the world transmits data in 8-bit byte chunks. Multicode's code points don't fit within an 8-bit byte, hence some form of transmission is a necessity.

Metacode facilitates the construction of higher-order protocols through the use of metadata. Each higher-order protocol defines its own tags using Metacode's metadata characters. Additionally, each higher-order protocol defines the meaning of their tags as well as the syntax for their use in Metacode streams. Moreover, we envision the creation of a common tag registration organization so that protocols may operate cooperatively.

Metacode provides precise definitions for data, metadata, protocols, and architectural layers. At its core, Metacode provides a mechanism for the unambiguous representation of textual content. Therefore, determining whether Metacode character streams are equivalent is also both precise, unambiguous and simple. The process of determining whether Metacode character streams are the same is the subject of section 6.3.2.

6.2 Metacode Compared to Unicode

In this section we relate Metacode's architecture to Unicode's organization. Specifically, we examine each layer of Metacode in detail making comparisons to Unicode when needed.

6.2.1 Transmission Layer

Metacode can utilize many of the popular binary character encoding transformation formats. We prefer UTF-32, UTF-16, and UTF-8. Which are the same transmission mechanisms that Unicode uses. All of these transmission mechanisms take as input integer based encodings and produce compressed binary sequences as output.

6.2.2 Code Point Layer

In Metacode code points would be specified by an integer, usually represented in hexadecimal. In the code point layer each code point would map to one and only one character. Additionally, each character would map to one and only one code point. In Metacode each code point would be of a fixed width. Code points in Metacode would never combine to form larger indices. In Metacode code points would generally be organized by script system. This is similar to Unicode. The most significant and novel feature of Metacode is a specific dedicated section of code points for the conveyance of meta information.

We considered two factors in attempting to find a suitable location for encoding Metacode's metadata characters. First, we wanted to select a range in Metacode that would allow for easy migration from Unicode. Second, we wanted to select a range that would allow Metacode's metadata characters to be simulated in other legacy encodings, in particular Unicode. We considered using Unicode's private use area, but ruled it out for two reasons: First, Metacode does not have any notion of a private use or user defined character area, which makes legacy conversion from Unicode more difficult. Second, the private use area within Unicode suffers from abuse due to the vast number of people using it conflicting ways.

We finally settled on using the surrogate range within Unicode. Unicode text processes already ignore this region, which permits simulation of metadata without disrupting metadata unaware processes, facilitating easy migration from Unicode to Metacode. Therefore, for purposes of demonstration the metadata code points are encoded in the following locations: 0xE0001, 0xE0002, and 0xE0020-0xE007F.

6.2.3 Character Layer

The character layer in Metacode is the place where the abstract data and metadata entities are defined. Additionally, we list the specific Unicode characters that would be excluded from Metacode as well as those legacy characters that would be redefined for Metacode. In cases where a Unicode character has no direct analog in Metacode we show how the same information can be expressed using Metacode's metadata characters.

6.2.3.1 Combining Characters

In Metacode each character is treated as an independent unit. Each Metacode character is unaffected by its surrounding neighbors. On the other hand, Unicode permits some neighboring characters to interact with one another to form new character units. Unicode refers to these character sequences as "combining characters". Metacode does not have any notion of combining characters at this layer. This sort of

interaction occurs within higher-order protocols. Therefore, we would not include any of Unicode's combining characters in Metacode. The Metacode stream on line 1 on Figure 6-2 illustrates the use of a higher-order protocol for the purpose of indicating combining characters.

Figure 6-2. Combining character protocol

ELM@CMBw¨**/ELM** — w diaeresis **(1)**

Table 6-1 lists the Unicode characters that would not be encoded in Metacode. The first column in Table 6-1 contains the excluded Unicode combining characters, the second column specifies the name of each character, while the third column contains the non-combining form for each character listed in the first column. [96]

In Metacode we would still like to be able to use the combining characters as pure data (content) minus their joining protocol, because in many instances these characters represent linguistic elements. To accomplish this we would redefine the combining characters for Metacode. In our redefinition these characters would be "pure", hence they would posses no special combining property. See Table 6-2 [96]. The first column in Table 6-2 lists the range of code points that would be redefined in Metacode. The second column list the type of characters that these code points represent.

In cases where a non-combining legacy character, mostly diacritic marks, already exists we would use it, rather than the newly redefined character. See Table 6-1. We take this approach for two reasons. First, if legacy Unicode text contained a diacritic character then there is a greater likelihood that they used the non-combining form of the character, because few text processes actually support com-

bining characters. Second, we reduce the number of redefined characters, thus easing migration from legacy Unicode to Metacode.

Table 6-1. Excluded Unicode combining characters

Code Point	Character Name	Non-Combining Code Point
U0300	GRAVE ACCENT	U0060
U0301	ACUTE ACCENT	U00B4
U0302	CIRCUMFLEX ACCENT	U005E
U0303	TILDE	U007E
U0304	MACRON	U00AF
U0306	BREVE	U02D8
U0307	DOT ABOVE	U02D9
U0308	DIAERESIS	U00A8
U0309	HOOK ABOVE	U02C0
U030A	RING ABOVE	U02DA
U030B	DOUBLE ACCUTE	U02DD
U030C	CARON	U02C7
U030D	VERITICAL LINE ABOVE	U02C8
U0310	CANDRABINDU	U0901
U0312	TURNED COMMA ABOVE	U02BB
U0313	COMMA ABOVE	U02BC
U0314	REVERSED COMMA ABOVE	U02BD
U0327	CEDILLA	U00B8
U0328	OGONEK	UO2DB

Table 6-2. Redefined Unicode combining characters

Code Point(s)	Character(s)
U0305, U030E-U030F, U0311, U0315-U0326, U0329-U0362	General diacritical marks
U0483-U0489	Cyrillic marks
U0591-U05AF	Hebrew cantillation marks

Table 6-2. Redefined Unicode combining characters (Continued)

Code Point(s)	Character(s)
U05B0-U05C4	Hebrew points and punctuation
U064B-U0652, U0670	Arabic points
U0653-U0655	Arabic maddah and hamza
U06D6-U06E9, U06EA-U06ED	Koranic annotation signs
U0711	SYRIAC SUPERSCRIPT LETTER ALAPH
U0730-U073F	Syriac vowels
U0740-U074A	Syriac marks
U07A6-U07B0	Thaana vowels
U0901-U0903, U093C, U093E, U094D, U0951-U0954	Devanagari signs
U093F-U094C,U0962-U0963	Devanagari vowels
U0981-U0983, U09CD, U09D7	Bengali signs
U09BE-U09CC, U09E2-U09E3	Bengali vowels
U0A02,U0A70-U0A71	Gurmukhi signs
U0A3E-U0A4D	Gurmukhi vowels
U0A81-U0A83, U0ABC, U0ACD	Gujarati signs
U0ABE-U0ACC	Gujarati vowels
U0B01-U0B03, U0B3C, U0B4D, U0B56-U0B57	Oriya signs
U0B3E-U0B4C	Oriya vowels
U0B82-U0B83, U0BCD, U0BD7	Tamil signs
U0BBE-U0BCC	Tamil vowels
U0C01-U0C03, U0C4D, U0C55-U0C56	Telugu signs
U0C3E-U0C4C	Telugu vowels
U0C82-U0C83, U0CCD, U0CD5-U0CD6	Kannada signs
U0CBE-U0CCC	Kannada vowels
U0D02-U0D03, U0D4D, U0D57	Malayalam signs
U0D3E-U0D4C	Malayalam vowels
U0D82-U0D83, U0DCA	Sinhala signs
U0DCF-U0DDF, U0DF2-U0DF3	Sinhala vowels
U0E30-U0E31, U0E34-U0E3A, U0E47	Thai vowels

Table 6-2. Redefined Unicode combining characters (Continued)

Code Point(s)	Character(s)
U0E48-U0E4B	Thai tone marks
U0E4C-U0E4E	Thai signs
U0EB1, U0EB4-U0EBB	Lao vowels
U0EBC, U0ECC-U0ECD	Lao signs
U0EC8-U0ECB	Lao tone marks
U0F35,U0F37, U0F39, U0F3E-U0F3F, U0F82-U0F84, U0F86-U0F87	Tibetan marks and signs
U0F71-U0F7D, U0F80-U0F81	Tibetan vowels
U0F7E-U0F7F	Tibetan vocalic modifiers
U0F90-U0FBC	Tibetan subjoined consonants
U102C-U1032	Myanmar vowels
U1036-U1039	Myanmar signs
U1056-U1059	Myanmar volcalic modifiers
U17B4-U17C5	Khmer vowels
U17C6-U17C8, U17CB-U17D3	Khmer signs
U17C9-U17CA	Khmer consonant shifters
U18A9	MONGOLIAN LETTER ALI GALI DAGALA
U20D0-U20E3	Diacritical marks for symbols
U302A-U302F	Ideographic diacritics
U3099-U309A	Hiragana voicing marks
UFE20-UFE23	Half marks

6.2.3.2 Glyph Variants

In Metacode there is no notion of glyph specific characters. This means that Metacode would not encode ligatures or specially shaped versions of characters. Therefore, Metacode would not incorporate any Unicode characters that are glyph composites or glyph variations of existing nominal characters. See Table 6-3 [96]. On the other hand, Metacode would provide a controlled mechanism for describing such information. In metacode, glyph variations would be specified by using

metadata tags. For example, the text stream on line 1 on Figure 6-3 demonstrates how a ligature protocol would be specified in Metacode.

Table 6-3. Unicode glyph composites

Code Point(s)	Description
U0132-U0133	Latin ligatures
U0587, UFB13-UFB17	Armenian ligatures
UFB00-UFB06	Latin ligatures
UFB1D-UFB4F	Hebrew presentation forms
UFB50-UFDFB	Arabic presentation forms (ligatures, initial, medial, final, isolated)
UFE30-UFE44	Glyphs for vertical presentation
UFE50-UFE6B	Small glyphs
UFE70-UFEFC	Additional Arabic presentation forms
UFF01-UFF5E	full-width Latin glyphs
UFF61-UFFEE	half-width Chinese, Japanese, and Korean

Figure 6-3. Ligature protocol

ELM@LIGﬂ**/ELM** — ﬂ ligature **(1)**

6.2.3.3 Control Codes

As we pointed out earlier some character encodings, like ASCII, have control codes. Metacode, however, by its nature does not need control codes. In this context we are referring specifically to ASCII's control codes. In Metacode legacy control codes would be captured by using metadata tags, just like any other higher-order protocol. Metacode does not prohibit other methods for expressing control codes. One such alternative method would be to create special singleton predefined metadata tags that would be directly encoded in Metacode, rather than being specified as an external higher-order protocol. This might ease migration from legacy encodings as well as reduce the number of characters in a stream. Another alternative would be to encode controls as normal Metacode characters. These characters would have no special semantics or required behavior. Text processes seeing these characters would

have the freedom to treat them just like any other character or to assign them special properties and behavior. It would be wise to limit these single meta characters otherwise we would just recreate all the problems with Unicode.

Occasionally there are some characters in legacy encodings that do not appear to be control codes, but behave as if they really are. For example, there are several characters encoded in Unicode for the purpose of indicating a break or space. See Table 6-4 [96]. The primary difference between each of these spacing characters is the amount of space to insert. For the most part these characters were encoded for historical reasons. Nonetheless, in Metacode we would not explicitly encode these characters. We would argue that the ideas expressed in these characters are best described as a higher-order protocol. In Metacode we would only encode a single space character. The single space character could be wrapped around a metadata tag that specified the exact amount of space to insert, which could be none. For example, the text stream on line 1 on Figure 6-4 demonstrates how a spacing protocol would be specified in Metacode. In order to aid comprehension the space character is represented by its code point value 0x0020.

Table 6-4. Unicode spacing characters

Code Point	Description
U2000	EN QUAD SPACE
U2001	EM QUAD SPACE
U2002	EN SPACE
U2003	EM SPACE
U2004	THREE PER EM SPACE
U2005	FOUR PER EM SPACE
U2006	SIX PER EM SPACE
U2007	FIGURE SPACE
U2008	PUNCTUATION SPACE
U2009	THIN SPACE
U200A	HAIR SPACE

Table 6-4. Unicode spacing characters (Continued)

Code Point	Description
U200B	ZERO WIDTH SPACE
U202F	NARROW NO BREAK SPACE
UFEFF	ZERO WIDTH NO BREAK SPACE

Figure 6-4. Spacing protocol

ELM@SP@EM0x0020**/ELM** — EM SPACE **(1)**

6.2.3.4 Metadata Tag Characters

In the Metacode system there would be 97 metadata characters (95 tag name characters and two tag protocol characters). The 95 tag name characters correspond to the ASCII and Unicode graphic character range 0x20-0x7E, while the tag protocol characters have no analog in either ASCII or Unicode. The two special tag protocol characters would be used for delimiting tags and for separating tag arguments. The base name of the tag characters are taken from Unicode. See Table 6-5. In Metacode, metadata characters would be used to specify higher-order protocols. The details of tag construction and protocol definition was discussed in the previous chapter.

Table 6-5. Metacode tag characters

ASCII Graphic Code Point	Character Name	Metacode Tag Character Name
41	LATIN CAPITAL LETTER A	TAG CAPITAL LETTER A
7A	LATIN SMALL LETTER Z	TAG SMALL LETTER Z
		TAG DELIMITER
		TAG ARGUMENT SEPARATOR

6.2.4 Character Property Layer

Each character in Metacode would be assigned properties. These properties are summarized in Table 6-6. The values for each character property, except for the

Metacode character name property appear in: Table 6-7, Table 6-8, Table 6-9, and Table 6-10. The Metacode character name property is an arbitrary sequence of alphabetic characters that represent the unique name of the character.

Table 6-6. Metacode character properties

Property	Description
Case	For those scripts that have case, indicates the specific case.
Script Direction	Indicates the preferred direction for a character to be written.
Code Point Value	Specifies the numeric index of a character.
Tag Character	Indicates whether a character is a data or a metadata.
Metacode Character Name	Represents the unique name of a character.

Table 6-7. Metacode case property values

Value	Description
U	Uppercase
L	Lowercase
T	Titlecase

Table 6-8. Metacode script direction property values

Value	Description
LTR	Left-to-right direction
RTL	Right-to-left direction
U	Unassigned, character is used in both left-to-right and right-to-left script systems.
I	Ignore, character is a metadata character and must be processed in logical order.

Table 6-9. Metacode code point value property

Value	Description
0xyyyyyyyy	32-bit hexadecimal index, where y is a hex digit

Table 6-10. Metacode tag property values

Value	Description
T	Character is a metadata character
F	Character is a data character

6.2.5 Tag Definition Layer

The tag definition layer is the layer of abstraction where Metacode higher-order protocols would be defined. The tag definition layer represents the core, most useful, and common protocols. See Table 6-11. Table 6-11 lists each protocol category along with an informative description.

In Metacode each higher-order protocol would remain distinct and separate from all others. This allows protocols to be interwoven without ill effect. Text processes would then be free to ignore specific protocols, if so desired. In some cases a process might not understand a protocol. In other cases a protocol might not be applicable to a particular process. In both situations a process can safely ignore such protocols. Above all, the Metacode protocol definition system is open ended allowing for ever more specialized protocols and private use protocols to be added.

The text on line 3 on Figure 6-5 illustrates how two protocols would be interwoven in Metacode. In this example we interleave the protocols appearing on lines 1 and 2 on Figure 6-5. The text on line 1 on Figure 6-5 illustrates how the higher-order collation protocol would be used to group characters together. In the collation protocol data characters surrounded by a collation tag would be treated as single unit for purposes of sorting. In general, text processes that perform sorting would make direct use of this information. The text on line 2 on Figure 6-5 illustrates how the direction protocol would be used communicate layout information to a process performing rendering. In the direction protocol characters surrounded by a direction tag would be rendered according to the specified direction. Nevertheless, when the two protocols are interwoven, the individual text processes would still be able to function properly. In this case the sorting process would ignore the direction tag and the rendering process would ignore the collation tag.

Figure 6-5. Interwoven protocols

ELM@COLLch**/ELM**ocolate (1)

DIR@Rchocolate**/DIR** (2)

DIR@R|ELM@COLLch**/ELM**ocolate**/DIR** (3)

Table 6-11. Major protocols

Category	Description
Language	The language of a stream or sub-stream of text.
Ligatures	A glyph representing two or more characters.
Collation	A text element containing two or more characters, that is treated as a single unit for purposes of sorting.
Presentation direction	The display order for a sequence of characters, horizontal and vertical.
Paragraphs	Characters that are used to indicate paragraph boundaries.
New lines	Characters that are used to indicate new line boundaries.
Combining characters	A character with one or more diacritics or vowels.
Glyph variants	Alternate character presentation forms, half-width or full-width.
Symmetric swapping	A character that when rendered uses its corresponding mirrored glyph rather than its normal glyph.
Transliteration	A text element contacting two or more characters, that is treated as a single unit for purposes of conversion. For example, case conversion.
General control codes	The C0 and C1 control codes. For example: line feed, tab, carriage return.
General layout controls	Typographic controls for spacing and line breaking.

6.2.6 Metacode Conversion

Yet another example of the power of the metadata mechanism is the ability to embed Unicode in Metacode. Conceivably the intersection of the Metacode and Unicode code points will not be the same and some legacy applications will require specific deprecated Unicode code points. Even this restricted case can be easily handled, because Metacode provides an open ended universal union of protocols. Because many of Metacode's data characters correspond directly to code points in Unicode round-trip conversion is easy. In Table 6-12 we provide sample mappings for those

Unicode characters that are deprecated in Metacode. In most situations the conversion of deprecated Unicode characters to Metacode protocols is obvious because the purpose of the Unicode characters is mapped to a specific Metacode protocol. Nevertheless, in some circumstances the intended use of a deprecated character may be impossible to determine from the data stream itself. In the case where the context is unknown, Metacode provides a “raw code point” protocol for preserving round-trip conversions.

The raw code point protocol works for embedding Unicode characters or any other character encoding protocol. For example, the Unicode code point U2007 (line 1 of Figure 6-6) this code point is designated as the figure space and is a hint about the presentation of spacing. We have a presentation spacing protocol and so this character is deprecated in Metacode. Suppose an application is looking for this specific character and has not fully migrated to Metacode. The raw code point protocol preserves this Unicode code point and any others. Line 2 on Figure 6-6 shows the embedding of U2007 **CP** stands for the raw code point protocol, **U** indicates the Unicode character set, and **2007** is the hexadecimal value of the Unicode code point. As always this protocol is preliminary and the details require further study and debate.

Figure 6-6. Metacode code point protocol

U2007 — figure space **(1)**

CP@U@2007 **(2)**

Table 6-12. Converting deprecated Unicode code points to Metacode

Category	**Description**	**Unicode**	**Metacode**
Language	Language Urdu	*<language tag>* *<tag letter u>* *<tag letter r>*	**LANG@UR**
		UE0001,UE0075,UE0072	ME004C,ME0041,ME004E, ME0047,ME0002,ME0075, ME0072

Table 6-12. Converting deprecated Unicode code points to Metacode (Continued)

Category	Description	Unicode	Metacode
Ligatures	Latin small ligature fi	fi UFB01	**ELM@LIG**fi**/ELM** ME0045,ME004C,ME004D, ME0002,ME0045,ME0049, ME0047,M0066,M0069, ME002F,ME0045,ME004C, ME004D
Presentation direction	Right-to-left embedding	*<RLE>* U202B	**DIR@R** ME0044,ME0049,ME0052,ME0002, ME0052
Paragraphs	Paragraph separator	*<p separator>* U2029	**PAR** ME0050,ME0041,ME0052
New lines	Line separator	*<l separator>* U2028	**BRK** ME0042,ME0052,ME004B
Combining characters	Latin small letter w with diaeresis	w¨ U0077,U0308	**ELM@CMB**w¨**/ELM** ME0045,ME004C,ME004D, ME0002,ME0043,ME004D, ME0042,M0077,M00A8, ME002F,ME0045,ME004C, ME004D
Glyph variants	Full-width Latin small letter a	a UFF41	**ELM@WID**a**/ELM** ME0045,ME004C,ME004D, ME0002,ME0057,ME0049, ME0044,M0061,ME002F, ME0045,ME004C,ME004D
Symmetric swapping	Activate mirroring	*<a symmetric s>* U206B	**MIR** ME004D,ME0049,ME0052

Table 6-12. Converting deprecated Unicode code points to Metacode (Continued)

Category	Description	Unicode	Metacode
General control codes	Carriage return	*<cr>* U000D	**N** M000D (singleton)
General layout controls	EM Space	*<em space>* U2003	**ELM@SP@EM /ELM** ME0045,ME004C,ME004D, ME0002,ME0053,ME0050, ME0002,ME0045,ME004D, M0020,ME002F,ME0045, ME004C,ME004D

In Metacode there is no limit to the number of protocols that can be expressed. Metacode can embed not only control and presentation protocols but also character coding standards, such as ISO-2022. In fact Metacode allows for a more natural expression of ISO-2022 escape sequences. For example, line 1 on Figure 6-7 shows the ISO-2022 byte sequence for announcing a switch to the ASCII character set. Line 2 shows how the announcement would be expressed in Metacode. The metadata sequence on Line 2 alleviates the requirement of having to decipher the escape sequence in order to determine the character set. In Metacode the escape sequence is replaced by the actual name of the character set. This approach offers the advantage that lookup tables and deciphering become unnecessary.

Figure 6-7. ISO-2022 escape sequence in Metacode

ESC (B **(1)**

ISO@ASCII **(2)**

In ISO-2022 the number of registered character sets is finite, because there are a fixed number of code points from which to make assignments. In Metacode the number of character sets that can be referenced is unlimited, because Metacode uses strings to encode names. This approach offers the greatest flexibility yet allows for

unambiguous communication. We suggest that the character set names be taken from the IANA (Internet Assigned Numbers Authority). The IANA records the names of character sets in RFC 1521.

6.2.7 Content Layer

The content layer is the highest layer of abstraction in Metacode's architecture. We anticipate that applications will primarily interact with Metacode at this layer of abstraction, as this layer deals with protocols and the raw content of Metacode streams. This is discussed in greater detail in the next section. This does not preclude a process from working with Metacode at some lower level of abstraction.

6.3 Data Equivalence

The concept of equivalent data streams is the subject of this section. In Metacode data steams may sometimes be considered to be equivalent even if they do not contain the same code points or bytes. Unicode also has a notion of equivalence, which they refer to as "normalization". We start this section by first examining in detail Unicode's normalization algorithm. We then describe Metacode's strategy for determining data equivalence.

6.3.1 Unicode Normalization

Normalization is the general process used to determine when two or more sequences of characters are equivalent. In this context the use of the term *equivalent* is unclear. It is possible to interpret the use of *equivalent* in multiple ways. For example it could mean characters are equivalent when their code points are identical, or characters are equivalent when they have indistinguishable visual renderings, or characters are equivalent when they represent the same content.

Figure 6-8. Non interacting diacritics

c¸^ U0063,U0327,U0302 **(1)**

c^¸ U0063,U0302,U0327 **(2)**

ḉ **(3)**

Unicode supports two broad types of character equivalence, canonical and compatibility. In canonical equivalence the term *equivalent* means character sequences that exhibit the same visual rendering. For example, the character sequences on lines 1 and 2 on Figure 6-8 both produce identical renderings, shown on line 3. [101]

In compatibility equivalence the term *equivalent* is taken to mean characters representing the same content. For example, line 1 on Figure 6-9 shows the single *ﬁ* ligature while line 2 on Figure 6-9 shows the compatible two character sequence *f* and *i*. In this case both sequences of characters represent the same semantic content. The only difference between the two is whether or not a ligature is used during rendering.[101]

Figure 6-9. Compatibility equivalence

ﬁ UFB01 **(1)**

fi U0066,U0069 **(2)**

6.3.1.1 Unicode Normal Forms

Unicode defines four specific forms of normalization based upon the general canonical and compatibility equivalences. These forms are listed on Table 6-13; the

title column indicates the name of the normal form, while the category column indicates the equivalence type. [101]

Table 6-13. Normalization forms

Title	Category	Description
Normalization Form D (NFD)	Canonical = visually equivalent	Canonical Decomposition
Normalization Form C (NFC)		Canonical Decomposition followed by Canonical Composition
Normalization Form KD (NFKD)	Compatibility = same content	Compatibility Decomposition
Normalization Form KC (NFKC)		Compatibility Decomposition followed by Canonical Composition

Normalization form NFD substitutes precomposed characters with their equivalent canonical sequence. Characters that are not precomposed are left as is. Diacritics (combining characters), however are subject to potential reordering. This reordering only occurs when sequences of diacritics that do not interact typographically are encountered, those that do interact are left alone. [101]

In Unicode each character is assigned to a combining class. Non Combining characters are assigned to the zero combining class, while combining characters are assigned a positive integer value. The reordering of combining characters operates according to the following three rules:

- Lookup the combining class for each character.
- For each pair of adjacent characters *AB*, if the combining class of *B* is not zero and the combining class of *A* is greater than the combining class of *B*, swap the characters.
- Repeat step 2 until no more exchanges can be made.

After all of the precomposed characters are replaced by their canonical equivalents and all non interacting combining characters have been reordered the sequence is then said to be in NFD.[96]

Normalization form NFC uses precomposed characters where possible, maintaining the distinction between characters that are compatibility equivalents. Most sequences of Unicode characters are already in NFC. To convert a sequence of characters into NFC the sequence is first placed into NFD. Each character is then examined to see if it should be replaced by a precomposed character according to the following rule.

- If the character can be combined with the last character whose combining class was zero, then replace the sequence with the appropriate precomposed character.

After all of the diacritics that can be combined with base characters are replaced by precomposed characters, the sequence is said to be in NFC. [101]

Normalization form NFKD replaces precomposed characters by sequences of combining characters and also replaces those characters that have compatibility mappings. In this normal form formatting distinctions may be lost. Additionally the ability to perform round trip conversion with legacy character encodings may be impossible, because of the loss of formatting. Normalization form NFKC replaces sequences of combining characters with their precomposed forms while also replacing characters that have compatibility mappings. [101]

There are some characters encoded in Unicode that need to be ignored during normalization. In particular, the bidirectional controls, the zero width joiner and non joiner. These characters are used as format effectors. The joiners can be used to promote or inhibit the formation of ligatures. Unicode does not provide definitive guidance as to when these characters can be safely ignored in normalization. Unicode only states these characters should be filtered out before storing or comparing programming language identifiers. [101]

To assist in the construction of the normal forms, Unicode maintains a data file listing each Unicode character along with any equivalent canonical or compatible mappings. Algorithms that wish to perform normalization must use this data file. By having all normalization algorithms rely on this data, the normal forms are guaranteed to remain stable over time. If this were not the case it would be necessary to communicate the version of the normalization algorithm along with the resultant normal form. [101]

6.3.1.2 Unicode Normalization Algorithm

The best way to illustrate the use of normal forms is through an example. Consider the general problem of searching for a string. In particular, assume that a text process is searching for the string “flambé”. Table 6-14 lists just some of the possible ways in which the string “flambé” could be represented in Unicode.

Table 6-14. The string “flambé”

#	Code Points	Description
1	U0066,U006C,U0061,U006D,U0062, U0065,U0301	decomposed
2	U0066,U006C,U0061,U006D,U0062, U00E9	precomposed
3	UFB02,U0061,U006D,U0062,U00E9	fl ligature, precomposed
4	UFB02,U0061,U006D,U0062,U0065, U0301	fl ligature, decomposed
5	UFF46,UFF4C,UFF41,UFF4D,UFF42, U00E9	full-width, precomposed
6	UFB02,UFF41,UFF4D,UFF42,U00E9	fl ligature, full-width, precomposed
7	U0066,U200C,U006C,U0061,U006D, U0062,U00E9	ligature supression, precomposed
8	U0066,U200C,U006C,U0061,U006D, U0062,U0065,U0301	ligature suppression, decomposed
9	U0066,U200D,U006C,U0061,U006D, U0062,U00E9	ligature promotion, precomposed

Table 6-14. The string “flambé” (Continued)

#	Code Points	Description
10	U0066,U200D,U006C,U0061,U006D, U0062,U0065,U0301	ligature promotion, decomposed
11	U202A,U0066,U006C,U0061,U006D, U0062,U00E9,U202C	left to right segment, precomposed
12	UFF46,U200C,UFF4C,UFF41,Uff4D, UFF42,U00E9	full-width, ligature promotion, precomposed
13	UFF46,U200D,UFF4C,UFF41,Uff4D, UFF42,U00E9	full-width, ligature suppression, precomposed

The character sequences found in Table 6-14 are all equivalent under the transforms NFKC and NFKD. In the case of NFKD, all transformations yield the sequence found on row 1 on Table 6-14, while transformations into NFKC result in the sequence on row 2 on Table 6-14. To demonstrate this, consider the conversion of Line 1 on Figure 6-10 copied from row 6 on Table 6-14 into NFKD. First, the sequence is converted to NFD by replacing precomposed characters with their decomposed equivalents. See line 2 Figure 6-10. Second, all characters that have compatibility mappings are then replaced by their corresponding compatibility characters. See line 3 Figure 6-10. The final sequence obtained is the same as the one found on row 1 of Table 6-14.

Figure 6-10. Conversion to NFKD

UFB02,UFF41,UFF4D,UFF42,U00E9 **(1)**

UFB02,UFF41,UFF4D,UFF42,U0065,U0301 **(2)**

U0066,U006C,U0061,U006D,U0062,U0065,U0301 **(3)**

The fact that all of the sequences found in Table 6-14 are equivalent under one normal form, in this case NFKD, does not necessarily mean that the sequences are equivalent in other normal forms. For example, consider line 1 on Figure 6-11 which is copied from row 3 in Table 6-14. When this sequence is converted to NFD

the result is line 2 on Figure 6-11. This does not match the sequence on row 1 of Table 6-14, therefore these sequences are not equivalent under NFD.

Figure 6-11. Conversion to NFD

UFB02,U0061,U006D,U0062,U00E9 **(1)**

UFB02,U0061,U006D,U0062,U0065,U0301 **(2)**

Thus far we have explored the details of data equivalence in Unicode. We now examine some of the problems that are caused by Unicode's normalization forms. In particular, we consider the interaction between the normalization process and other Unicode algorithms.

6.3.1.3 Problems with Unicode Normalization

The overall complexity of normalization presents serious problems for general searching and pattern matching. Without a single normalization form, it is not possible to determine reliably whether or not two strings are identical. The W3C (World Wide Web Consortium) has advocated adopting Unicode's NFC for use on the web. Additionally, W3C recommends that normalization be performed early (by the sender) rather than late (by the recipient). Their recommendation is to be conservative in what you send, while being liberal in what you accept. The major arguments for taking this approach are: [107]

- Almost all data on the web is already in NFC.
- Most receiving components assume early normalization.
- Not all components that perform string matching can be expected to do normalization.

There are some problems with this strategy, however. It assumes that the primary purpose of normalization is to determine whether two sequences have identical renderings which is appropriate for display but inappropriate for information processing. In Unicode's NFC any and all formatting information is retained. This causes

problems for those processes that require comparisons to be based only upon raw content, such as web search engines and web based databases. Additionally, it places undo limitations on the characters that can be used during interchange.

During the process of normalization the properties of the characters are not guaranteed to remain stable. In Unicode the *numero sign* is a neutral character, while the *latin capital letter n* and *latin small letter o* are left-to-right characters. Obviously these character types are not the same. Therefore, it is no surprise that when the run on line 3 on is converted to display order it does not match the unnormalized display order. See lines 4 and 2 respectively. This example reveals Unicode's strong ties to presentation rather than content. This might seem unimportant, as the visual display is not vastly different. It could lead to cases in which incorrect conclusions could be drawn. See Figure 6-13.

Figure 6-12. Protocol interaction

U0627,U2116,U0031,U0032,U0033 **(1)**

1 2 3 № ا **(2)**

U0627,U004E,U006F,U0031,U0032,U0033 **(3)**

N o 1 2 3 ا **(4)**

Line 1 on Figure 6-13 is a run of characters in logical order with its corresponding display order on line 2 on Figure 6-13. The run on line 3 on Figure 6-13 is the normalized form of line 1 on Figure 6-13, with its display order on line 4 on Figure 6-13. When the two display orderings are compared the results are radically different visually and semantically. See lines 2 and 4. These examples further illustrate the need for a new data encoding model.

Figure 6-13. Data mangling

U0627,U00BC **(1)**

¼ ا **(2)**

U0627,U0031,U2044,U0034 **(3)**

4 / 1 ا **(4)**

The next serious problem with the normalization process is the unexpected interaction with other Unicode protocols. In particular, the Unicode Bidirectional Algorithm. The run of characters on line 1 on Figure 6-12 is a sequence of Arabic characters in logical order. The text displayed on line 2 on Figure 6-12 is the same run of characters but in display order. This is the output from the Unicode Bidirectional Algorithm.

When the run of characters on line 1 on Figure 6-12 are placed into NFKC the result on line 3 on Figure 6-12 is obtained. Placing the run of characters on line 1 on Figure 6-12 into NFKC causes the *numero sign* (U2116) to be converted to the two character sequence, *latin capital letter n* (U004E) followed by *latin small letter o* (U006F). This illustrates the unanticipated and confounding interaction between normalization and bidirectional processing.

In the previous examples we have seen some of the unexpected interactions between normalization and layout. In the next section we explore data equivalence in Metacode. In particular, we will see that in Metacode data equivalence does not interact with other algorithms. This is possible, because in Metacode data equivalence is steered away from presentation and redirected towards content.

6.3.2 Data Equivalence in Metacode

In Unicode the definition of equivalence is strongly tied to the visual appearance of characters. In the Metacode system, however equivalence is aligned towards the meaning of characters, rather than their visual representation. Therefore, in Metacode we would define three types of data equivalence:

- Byte equivalence — If two streams contain the same sequence of bytes then the two streams are said to be byte equivalent. See the Haskell function `byteEquivalent` in Appendix D.
- Code point equivalence — If two streams contain the same sequence of code points (data and metadata) then the two streams are said to be code point equivalent. See the Haskell function `codePointEquivalent` in Appendix D.
- Content equivalence — If two streams contain the same sequence of data code points (excluding metadata) then the two streams are said to be content equivalent. See the Haskell function `contentEquivalent` in Appendix D.

In Metacode each form of data equivalence is linked to a specific architectural layer within Metacode (starting from the lowest layer of abstraction):

- Transmission layer — Byte equivalence
- Code point layer — Code point equivalence
- Content layer — Content equivalence

Equivalence at a lower layer of abstraction guarantees equivalence at higher layers. This is summarized in the following three rules:

- If two streams are byte equivalent then the two streams must also be code point equivalent and content equivalent.
- If two streams are code point equivalent then the two streams must also be content equivalent. The two streams, however may optionally be byte equivalent.
- If two streams are content equivalent the two streams may optionally be code point equivalent and or byte equivalent.

Looking at Unicode’s normalization algorithm we find it to be very complex with ill defined boundaries. Metacode’s content equivalence however, is simple with well defined boundaries. Metacode’s content equivalence is performed by simply

comparing the data characters in a stream allowing metadata characters to be completely ignored. This is possible because metadata characters do not play any role in determining content. The metadata characters always express higher-order protocols and have no effect on the interpretation of the raw data.

Here is a list of the other properties of Metacode's equivalence algorithm:

- The algorithm is reversible.
- The algorithm is robust, it still functions as new tags are created.
- The algorithm is applicable to all text processes, it assumes no particular type of text process whether it be presentation or content based.
- The algorithm is independent and separate from other algorithms

Unicode's normalization algorithm does not exhibit these properties. We argue that the definition of Metacode content equivalence is in fact what Unicode normalization should have been. Metacode's definition of content equivalence is more closely aligned with Unicode's goal of separating characters from glyphs than Unicode normalization is. Unicode normalization should not have been concerned with how particular characters are rendered. In Metacode this would be the privy of rendering engines.

In Metacode there is no limitation on which characters can and cannot be used in Metacode's content layer. Unicode by contrast, has limitations regarding the characters that can and cannot be used in Unicode's normal forms. Metacode's approach allows text components to be both liberal in what they send and receive. The presence of metadata in a stream never alters the interpretation of the raw content.

Next we revisit the problem of expressing the string "flambé". See Table 6-14. In this table we concluded that all the entries were equivalent. Table 6-15 captures the same semantics using Metacode's metadata characters as was expressed in Table 6-14. To enhance comprehension of the table we use the printed version of the characters, rather than their code point values. Additionally, in Table 6-15 we do not

find any combining characters, because this notion is only applicable to Unicode and is unnecessary in Metacode. We refer to the strings in Table 6-15 as being content equivalent, that is they all represent the same raw content. Moreover, counting the number of entries in Table 6-15 we find that this is less than half the number of entries in Table 6-14. This is not surprising given the numerous ways in which the same content can be expressed in Unicode. In Metacode, however there would never be a case where the same raw content could be expressed in more than one way.

Table 6-15. The string "flambé" in Metacode

#	String	Description
1	flambé	no higher-order protocols
2	**ELM@LIG**fl**/ELM**ambé	fl ligature protocol
3	**ELM@WID**flambé**/ELM**	full-width protocol
4	**ELM@WID|ELM@LIG**fl**/ELM**ambé**/ELM**	fl ligature and full-width protocols
5	**DIR@L**flambé**/DIR**	direction protocol

In Metacode, data equivalence is never based on any external tables, thereby eliminating any potential data table versioning problems. For example, consider Unicode character U2048, *question exclamation mark ?!*. See line 1 on Figure 6-14. This character was first defined in Unicode 3.0. The purpose of the character is to produce a special glyph variant of a *question mark ?* combined with an *exclamation mark !* for use in vertical writing systems. See line 2 on Figure 6-14. Nonetheless, the meaning of the *question exclamation mark* is the same as the combined individual characters. This required Unicode to update their normalization tables. Nevertheless, when applications based on earlier versions of Unicode performed normalization the *question exclamation mark* would not match the individual *question mark* and *exclamation mark*. Therefore, these characters would be incompatible.

Figure 6-14. Question Exclamation Mark

U2048 ?! **(1)**

U003F,U0021 ?! **(2)**

Using Metacode and metadata tags no such dependency on a specific version of Metacode is necessary. In Metacode a new code point definition would not be required at all. This vertical form using metadata is illustrated on line 2 on Figure 6-15. When line 2 is compared to line 1 we find the two are content equivalent; both strings represent the same content. If at some later time we find it necessary to add a wide form of the *question exclamation mark* to Metacode we need only surround the *?* and *!* with a metadata tag. See line 3 on Figure 6-15. Thus, Metacode and its associated metadata tagging mechanism is both open and flexible. The process for determining whether the two streams are content equivalent does not require any changes to accommodate the use of this tag further illustrating the openness of the Metacode architecture.

Figure 6-15. Metadata Question Exclamation Mark

?! **(1)**

ELM@VER?!**/ELM** **(2)**

ELM@WID?!**/ELM** **(3)**

6.3.3 Simulating Unicode in Metacode

Metacode permits the simulation of Unicode and its normalization forms easing migration to our new architecture. Unicode's normalization forms would be encoded by using Metacode's metadata tagging mechanism. The notion of an Unicode combining character would be described as a higher-order protocol, previously illustrated on Figure 6-2. Unicode's normalization algorithm would be yet another form of data equivalence. For example, the Metacode character stream on line 1 on Figure 6-16 would represent the Unicode characters *latin capital letter u* and

combining diaeresis. Line 2 on Figure 6-16 is the single Metacode character *latin capital letter u diaeresis*. In Metacode the streams on lines 1 and 2 would be unequal, because they do not represent the same content. Nevertheless, the two streams would be equivalent under Unicode simulation, because the streams have identical renderings.

Unicode normalization can be thought of as a higher-order form of data equivalence. We call this form of equivalence “display equivalence” and place it in a higher layer over content equivalence. Display equivalence does not violate any of our earlier rules of equivalence.

Figure 6-16. Simulating Unicode normalization

ELM@CMBU¨**/ELM**	**(1)**
Ü	**(2)**

6.4 Code Points vs. Metadata

Metacode is capable of easy expansion to accommodate the inevitable and boundless growth in written expression. In Metacode expansion may occur in both the code point layer and in the tag definition layer. Our architecture encourages relatively infrequent expansion at the code point layer when a new natural language construct needs to be expressed. Expansion at the tag definition layer would occur only when information describing a natural language construct needed to be expressed. Our architecture greatly reduces the number of instances where appropriate assignment is ambiguous. For the remainder we improve the situation by providing more workable options for capturing the essence of natural language constructs.

6.4.1 Metacode Principles

In many cases the decision as to whether to use a code point or a tag would be obvious. Nevertheless, some heuristics would be established to provide guidance

with these decisions. For example, the following heuristics are indicators for encoding a concept as a code point:

- The concept represents a natural language construct.
- The concept represents a fundamental element of some formal system.

The following list are indicators for encoding concepts as protocols, rather than as code points.

- The concept is a stylistic variation of an already existing code point.
- The concept is used for signaling or control of some higher-order process.
- The concept causes the semantics of code points to change.
- The concept provides meta information about an existing code point.
- The concept is a specialization or generalization of an already existing tag.

6.4.2 Applying Metacode Heuristics

Bellow we illustrate how these Metacode heuristics would be applied to encoding new objects within Metacode. We examine situations in which the decision is easy to make as well as those in which the decision is less clear.

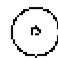
6.4.2.1 Natural language text

First, we explore the case where the decision as to whether to use a code point or a tag is unambiguous. Let us look at the task of encoding Egyptian Hieroglyphic symbols in Metacode. The Egyptian hieroglyphic symbols are divided into two classes, phonograms and ideographs. Phonograms are used to write the sounds of the language. The value of the sound was usually obtained from the name of the object being depicted. The hieroglyphic symbol “foot” on Figure 6-17 represents a consonant that is pronounced as the letter “b” in English. As the “foot” object is both an element of a natural language (Egyptian) and an element of a formal system (hieroglyphics). It would be encoded as a code point. [17], [80]

Figure 6-17. Egyptian hieroglyphic phonogram

In Egyptian, ideographs represent either the actual object being depicted or a closely related idea. For example, consider the hieroglyphic symbol “ra” on Figure 6-18. The symbol ra stands for the sun. Even though this is not a phonogram, the ra symbol is an element of written natural language and would also be encoded as a code point.

Figure 6-18. Egyptian hieroglyphic ideograph



6.4.2.2 Mathematics

There currently exists several systems for representing mathematical documents, such as TeX, Mathematica, and MathML [65]. These systems, however, deal with the representation of mathematics at the document level and not at the character level. Unicode has recently taken steps to fill this gap by encoding a set of characters within the surrogate range specifically designed for mathematics. These mathematical characters include bold, italic and script Latin letters, bold and italic Greek letters, and bold and italic European numerals [98]. Metacode would not include such characters because such information is captured by Metacode’s higher-order protocols. Unicode’s mathematical characters are really stylistic variations of already encoded characters. On the other hand, it is true that the semantics of the mathematical characters differ from the basic Latin, Greek, and European numerals. In Metacode we would not prevent such semantics from being expressed, rather we

would argue that the use of code points as the means for their expression is incorrect because these characters are stylistic variations of already encoded characters.

In the case of Unicode we find that only the European numerals have bold forms, what about all the other types of digits? If we believe that using code points is the correct approach, then we must be prepared to provide multiple forms of every kind of digit (Arabic, Chinese, Japanese, etc.) Moreover, the introduction of multiple stylistic forms of letters and digits only serves to make data equivalence more complex.

In Metacode the semantics of mathematics are expressed through a higher-order protocol. For example, rather than encoding bold characters Metacode would express such information by using a tag. See line 1 on Figure 6-19. On line 1 on Figure 6-19 we see the *latin capital letter a* surrounded by a *math* tag with the single argument *bold*. Moreover, in Metacode the introduction of mathematical tags does not require any changes to our data equivalence procedures.

Figure 6-19. Mathematical characters

MATH@BA/MATH **(1)**

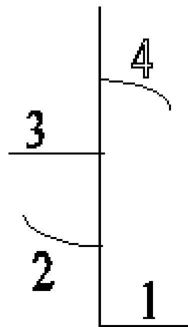
6.4.2.3 Dance notation

Metacode is not limited to just encoding natural languages and mathematics, although we anticipate that these will be the predominant uses. It is possible to use Metacode to encode other formal systems. For example, consider the system for expressing dance movements. This is a formal system and is called "Action Stroke Dance Notation" (ASDN). ASDN is a movement shorthand designed to capture the basic movements of dance as actions and strokes. In ASDN each of the basic actions is represented using a graphic symbol, known as an "action stroke". Each of these actions indicates a movement of either the leg or arm staffs. Each action stroke is attached to a vertical line called the staffline. Lines written to the left of the staffline

signal a movement of a left limb, while lines written to the right signal a movement of a right limb. [19]

In the illustration on Figure 6-20, #1 is a step with the right leg, #2 is an air gesture with the left leg, #3 is a step with the left leg, and #4 is a touch gesture with the right leg. In Metacode we would encode each of these symbols as individual code points, because they represent the fundamental elements of ASDN and are part of a formal written system. [19]

Figure 6-20. Action Stroke Dance Notation



In ASDN the direction of movement (forward and backward) can also be expressed. In ASDN these are represented by using up and down arrows. In Metacode we would not encode these movement symbols, because these objects are already encoded in Metacode, albeit not as part of ASDN notation. The movement symbols are combined with action strokes to indicate the direction of a stroke. For example, the action stroke on Figure 6-21 indicates a backward movement of the right leg. [19]

Figure 6-21. Action Stroke Dance Notation with movement

In Metacode we could express the combination of an action stroke and direction in two ways. First, we could encode each combination of an action stroke and direction as an individual code point. Second, we could create a tag for combining a movement direction with an action stroke. We would argue that the second approach is more appropriate, because the combination of an action stroke and a direction does not represent a fundamental unit of the formal system. The combination of an action stroke and a direction is a composite object constructed from two fundamental elements. Therefore, we would describe the composite object by using a higher-order protocol "tag". For example, in Metacode we would capture the semantics of Figure 6-21 by using the Metacode character sequence on line 1 on Figure 6-22. To simplify comprehension of Figure 6-22 the action stroke and movement direction are specified using their long name (italic characters), rather than by their individual code points.

Figure 6-22. Metacode Action Stroke Dance Notation tag

ASDN*right-step, down-arrow***/ASDN** **(1)**

6.5 Benefits of Metacode

We argue that multilingual character coding systems should provide both a set of unambiguous data characters and a mechanism for specifying meta information about those characters. In Metacode characters are identified by their meaning rather than by their shape. Additionally, Metacode's "data characters" are always distinct and separate from "metadata characters". Metacode's open ended tag mechanism allows for the definition of an unlimited number of possible protocols, yet does not require any future code points. By adopting this framework Metacode is free to deal

entirely with the definition of characters. This approach affords the greatest level of flexibility, while still retaining the ability to process multilingual data efficiently.

Metacode does not dictate how metadata should be used. Metacode solely deals with mechanism. The semantics of protocols are left to higher-order processes. Metacode gets to separate protocol definition from character picking. The once fuzzy boundary separating characters from protocol is now replaced by a well defined border. This precise separation greatly simplifies the construction of multilingual information processing applications.

7 Conclusions

In Metacode information processing is focussed on content and away from display. In our architecture roles and responsibilities are clearly indicated. We have sharpened the focus on the indistinct boundary separating code points, characters, and control information. The tasks of assigning code points and defining protocols are now separate and distinct activities. This separation of activities promotes the deprecation of code points that convey control information. In particular, ligatures, control codes, glyph variants and half-width forms. In Metacode control information is captured by the metadata layer, irrespective of whether the control relates to presentation or content.

7.1 Summary

In Chapter 2, we outlined the overall field of software globalization. We provided definitions for the terms internationalization, localization, and translation. We examined several challenges to creating multilingual software. We described in detail each of the six subfields of software globalization: Translation, International User Interfaces, Cultural/Linguistic Formatting, Keyboard Input, Fonts, and Character Coding Systems.

In Chapter 3, we discussed character sets and character coding systems. We defined the relevant terms related to character coding systems. We examined in detail several character coding schemes, covering both fixed-width stateless and variable-width stateful systems. We made arguments justifying the need for multilingual character coding systems. We described in detail four multilingual coding systems:

Unicode, Multicode, TRON, and EPICIST. We concluded that Unicode's fixed width stateless encoding system is a good mechanism for encoding/transmitting code points.

In Chapter 4, we considered problems arising from using multilingual character encodings. Specifically, we examined the problem of processing bidirectional scripts (Arabic, Hebrew, Farsi, and Urdu). Initially, we considered the proposition that the processing of bidirectional text was an algorithmic problem. We explored several algorithms for processing bidirectional text: Unicode Bidirectional Algorithm, FriBidi, PGBA, ICU, and Java, but found them inadequate. We created a functional bidirectional algorithm (HaBi), because a functional implementation would enable us to discover the true nature of bidirectional text processing. We found our HaBi implementation incomplete, however. We concluded that the bidirectional processing problem was not an algorithmic problem but an architectural problem. The existence of this architectural problem points to fundamental flaws in the underlying character set centric model.

In Chapter 5, we explored several strategies for addressing the shortcomings of the character set centric model. In particular, we looked at using metadata for describing more of the underlying structure of scripts. We examined XML as a metadata model for multilingual information processing, but found it inappropriate. We defined our own general metadata model, presenting evidence of its suitability for multilingual information processing. We introduced several meta tags (text element, direction, mirroring, and language) that showed how complex semantics could be captured in our metadata model. We established that both XML and HTML could be captured using our general metadata model. Applying our metadata model to the bidirectional text processing problem enabled us to discover the true nature of bidirectional text processing (inferencing and reordering). We concluded that our metadata model allowed for a general reorganization of multilingual information processing.

In Chapter 6 we developed our multilingual information processing architecture. Our architecture incorporates character sets, metadata, and core protocols, providing an overall framework for multilingual information processing. We named our architecture Metacode to reflect our focus on higher-order protocols. We established that it is easy to migrate to Metacode because Unicode can be simulated in Metacode. We eliminated the need for complex normalization algorithms by introducing a hierarchy of simple data equivalences (byte, code point, content). We concluded by summarizing the benefits our architecture provides.

7.2 Contributions

In this dissertation we made practical contributions to several issues in multilingual information processing. These contributions emerged from the study of four areas: bidirectional processing, normalization, characters, and higher-order protocols.

In summary the contributions of this dissertation are:

- We developed an architecture that unambiguously separates code points, content, control information, and display.
- We created an architecture that minimizes harmful interactions.
- We created an extendable metadata mechanism for describing higher-order protocols.
- We conducted a detailed analysis of bidirectional reordering algorithms and discovered the essence of bidirectional processing.
- We established that our architecture allows for a separation of bidirectional inferencing from bidirectional reordering.
- The metadata architecture that we developed supports reversible and detectable information processing algorithms.
- We created a hierarchy of fundamental data equivalences that are simple to implement.
- We showed that data equivalence algorithms can operate independently from any particular version of Metacode.

- We demonstrated that it is possible to simulate Unicode in our Metacode system, allowing for easy migration.
- We provide guidance for encoding concepts in Metacode.

In the paragraphs below we discuss each of these contributions.

We presented evidence that supports our argument that there is little separation between content, display, and control information. We showed several instances (e.g., ligatures, half-width forms, and final forms) where it was laborious to separate display information from content. In many places throughout the dissertation we showed the difficulty of separating control information (e.g., bidirectional controls and breaking controls) from content. We developed abstractions and mechanisms for better delineating the boundaries between control information, display and content.

In our metadata and protocol abstractions we showed that intra-layer change was minimized as we encoded new concepts. In the character set centric view, change cannot be localized (e.g., spacing code points, see section 6.2.3.3). This shift in philosophy is significant because it prevents unanticipated interactions as we saw in the case of normalization. We developed a spacing protocol for expressing variable length spaces that did not require any change in the code point layer. As we defined other protocols in the tag definition layer (e.g., combining characters and direction) we showed that the content layer was unaffected.

In the character set centric approach endless code point tinkering is required each time a new linguistic or cultural variation is added (e.g., Arabic presentation forms). This strategy is limiting because there are only a finite number of code points for encoding characters. We demonstrated that in Metacode linguistic and cultural variations are captured by the open-ended metadata definition system. We saw that Metacode's universal text element protocol captured a wide array of linguistic and cultural information (e.g., glyph variants and final forms, see section 5.6) yet did not require additional code points.

In this dissertation we showed that the character set centric position does not supply the best set of basic building blocks for constructing higher-order protocols. This is important because it leads to the use of overloaded characters and conflicting solutions as we saw in the case of XML. In Metacode we provide a better foundation for higher-order protocols. We showed that our metadata system enabled a more cohesive form of XML (see section 5.4), because we avoid the problems of entity references.

Our analysis of bidirectional algorithms revealed the true nature of bidirectional information processing. This is significant because we have separated the linguistic processing from the information processing. We concluded that bidirectional processing is comprised of two activities: inferencing and reordering (see sections 4.8 and 5.7). Inferencing takes natural language text in its most primitive and basic form and inserts cultural and linguistic assumptions (e.g., language and direction of script) into the stream. Reordering converts attributed natural language text into a form that is suitable for presentation.

We presented evidence that current bidirectional reordering algorithms fail to separate the activities of inferencing and reordering as we saw in the case of Unicode and FriBidi. This lack of separation causes bidirectional algorithms to generate inappropriate output, code points in display order. We argue that only inferencing is appropriate in the context of character coding systems. Reordering is an activity that should occur in higher-order processes. In our bidirectional algorithm (HaBi) we separate inferencing and reordering, always keeping data in logical order.

We presented evidence that showed that the effects of bidirectional processing were both difficult to detect and reverse. This is significant because in many cases it is necessary to undo bidirectional processing as we saw in the case of domain names. We found that without separating inferencing from reordering the algorithm converting from display order to logical order was not a one-to-one function. The

well known bidirectional algorithms are not reversible. There are no identifying markers in the processed text leading to confusion over whether a text stream was processed. We demonstrated that the effects of our bidirectional algorithm are reversible and detectable.

We demonstrated that there are unexpected and damaging interactions between normalization and bidirectional processing as we saw in the case of fractions and Arabic text (see section 6.3.1.3). We argued that these interactions point to erroneous assumptions about the role of normalization in character coding systems. Normalization of presentation forms cannot be solved at the code point layer. This is crucial because the Metacode approach frees information processing from presentation issues. Current character coding systems assume that the purpose of normalization is to determine if characters look the same. In the Metacode system normalization is never based on the visual appearance of characters, but rather on the underlying abstract meaning of the characters. This allows protocols to coexist without interference as we saw in the case of bidirectional processing and normalization.

We presented evidence that in the character set centric model normalization algorithms must be rewritten each time a new display variation is added, because the only mechanism for expressing display variation is code points. We showed that Unicode's normalization tables required updating when they added a vertical variant of the question exclamation mark character (see section 6.3.2). In our Metacode system we have options, we could use code points or metadata protocols. We encoded the vertical question exclamation mark using our universal text element protocol. This approach minimizes the need to rewrite normalization algorithms each time a new display variant is introduced.

We showed that Metacode's data equivalence algorithms (byte, code point, and content) operate independently from the visual appearance of characters. This is

significant because it allows construction of data equivalence algorithms that do not require change as new protocols and code points are defined.

We presented evidence that Unicode data can be easily converted to Metacode without a loss of semantics which we saw in the conversion of combining characters and control codes (see section 6.2.6). We showed that Unicode's normalization algorithm could be simulated in Metacode. Both of these are important because an easy migration path is necessary to encourage use of Metacode.

Throughout this dissertation we offered guidance and examples for encoding concepts in Metacode. We studied several examples from Hieroglyphics, mathematical typesetting, and Action Stroke Dance Notation to illustrate the correct use of the architecture. We demonstrated that in many cases it was easy to decide whether to use a code point or a protocol. In other cases we found the decision to be more complicated as we saw in Action Stroke Dance Notation.

7.3 Limitations

In our Metacode system we made some trade-offs in order to achieve greater functionality. These trade-offs are summarized as follows:

- Data encoded in Metacode in some cases require more memory than other multilingual encodings.
- Metacode data in some situations takes longer to transmit than other multilingual encodings, because character streams may use more memory.
- In Metacode the storage unit (character) is no longer equivalent to the logical unit (text element), making manipulation of data more complex.
- Editing of Metacode data is more elaborate.

7.4 Future Work

In this dissertation we provide only a small number of meta tags. Further research and study of higher-order protocols, display hinting and linguistic elements would be need in a full implementation of our architecture. In the discussion of our

architecture we did not specifically focus on the actual characters that would be encoded in Metacode. In a complete implementation the actual choice of characters would be necessary. This character assignment activity would have to consider issues related to combining characters, Han unification, and control codes. Additional study and debate is needed.

In some cases it might be desirable to keep some number of controls (e.g., new line, and bell) as singleton code points to ease migration. In other cases it may be more advantageous to deprecate controls (e.g., right-to-left mark) and recast them as higher-order protocols to avoid the trouble they cause. This issue would require careful consideration and debate.

We anticipate that frequently used higher-order protocols (e.g., ligature) will need to have shorthand or singleton representations to minimize memory utilization. This activity would require identification of the commonly used protocols and discussion over which protocols would benefit the most from using an abbreviated form.

References

[1] Abed, Farough. "Cultural Influences on Visual Scanning Patterns." *Journal of Cross-Cultural Psychology,* December 1991, pp 525-535.

[2] Abramson, Dean. "Optimized Implementations of Bidirectional Text Layout and Bidirectional Caret Movement." *13th International Unicode Conference,* September 1998.

[3] Adams, Glen. "Internationalization and Character Set Standards." *Standard View, The ACM Journal on Standardization,* Volume 1, 1993.

[4] Alhadif, Mohamed. "International Music Festival." *Alshafha,* 10 July 2001, p 3. (in Arabic)

[5] Alvestrand, Harald Tevit. "IETF Policy on Character Sets and Languages." RFC 1766, March 1995.

[6] Apple Computer. *Inside Macintosh Text.* Addison-Wesley. 1993.

[7] Apple Computer. "About Apple Advanced Typography Fonts." February 1998.

[8] Atkin, Steven. "A Dynamic Object-Oriented Approach to Software Internationalization." *Master's Thesis Florida Institute of Technology,* December 1994.

[9] Atkin, Steven and Borgendale, Ken. "IBM Graphical Locale Builder." *12th International Unicode Conference,* April 1998.

[10] Atkin, Steven and Stansifer, Ryan. "Implementations of Bidirectional Reordering Algorithms." *18th International Unicode Conference,* April 2001.

[11] Au, Sunny. “Hello, World! A Guide For Transmitting Multilingual Electronic Mail.” *Proceedings of the 23rd ACM SIGUCCS conference on Winning the networking game,* October 1995, pp 35-39.

[12] Becker, Joseph. “Arabic Word Processing.” *Communications of the ACM,* July 1987, Volume 30, Number 7, pp 600-610.

[13] Becker, Joseph. “Unicode 88.” *Xerox Corporation,* 1988.

[14] Belge, Matt. “The Next Step In Software Internationalization.” *Interactions,* January 1995, Volume 2, Number 1, pp 21-25.

[15] Bemer, R. W. “The American Standard Code For Information Interchange.” *Datamation,* 9, No. 8, 32-36, August 1963, and ibid 9, No. 9, 39-44, September 1963.

[16] Bettels, Jürgen and Bishop, Avery F. “Unicode: A Universal Character Code.” *Digital Technical Journal,* 1993, Number 3, Volume 5, pp 21-31.

[17] Budge, E.A. Wallis. *An Egyptian Hieroglyphic Dictionary.* Dover Publications. 1978.

[18] Clark, James. “Minority WG Opinion on XML C14N and Unicode C14N.” Available: *http://www19.w3.org/Archives/Public/www-xml-canonicalization-comments/2000Jan/0000.html.* Retrieved: January 21, 2001.

[19] Cooper, Iver P. “Action Stroke Dance Notation.” Available: *http://www.geocites.com/Broadway/Stage/2806/.* Retrieved: August 12, 2001.

[20] Davis, Mark. et al. “Creating Global Software: Text Handling and Localization in Taligent’s CommonPoint Application System.” *IBM Systems Journal,* 1996, Number 2, Volume 35, pp 227-242.

[21] Davis, Mark. et al. “International Text In JDK 1.2.” Available: *http://www.ibm.com/java/education/international-text/*. Retrieved: July 17, 2000.

[22] Democratic Peoples Republic of Korea. “DPRK Standard Korean Graphic Character Set for Information Interchange.” KPS 9566-97, 1997.

[23] Dürst, Martin and Freytag, Asmus “Unicode in XML and Other Markup Languages.” Available: *http://www.unicode.org/unicode/reports/tr20.* Retrieved: January 9, 2001.

[24] Edberg, Peter. “Tutorial: Survey of Character Encodings.” *11th International Unicode Conference,* September 1997.

[25] Erickson, Thomas D. “Working With Interface Metaphors.” in *The Art of Human Computer Interface Design.* edited by Brenda Laurel, Addison-Wesley. 1990.

[26] European Computer Manufacturers Association. “7-Bit Coded Character Set.” ECMA-6, December 1991.

[27] European Computer Manufacturers Association. “Character Code Structure and Extension Techniques.” ECMA-35, December 1994.

[28] Fateman, Richard “Algol 60, a language, a report.” Available: *http://www.cs.berkley.edu/~fateman/264/lec/notes17.pdf*. Retrieved: April 17, 2001.

[29] Fernandes, Tony. *Global Interface Design.* AP Professional. 1995.

[30] Flanagan, David. *Java in a Nutshell*. O’Reilly and Associates. 1999.

[31] Goundry, Norman “Why Unicode Won’t Work On The Internet: Linguistic, Political, and Technical Limitations.” Available: *http://www.hastingsresearch.com/net/04-unicode-limitations.shtml.* Retrieved: June 5, 2001.

[32] Graham, Tony. *Unicode A Primer.* M&T Books. 2000.

[33] Graham, Tony. “Changes in Unicode that led to changes in XML 1.0 Second Edition.” Available: *http://www-106.ibm.com/developerworks/library/u-xml.html.* Retrieved: January 20, 2001.

[34] Grobgeld, Dov. “A Free Implementation of the Unicode Bidi Algorithm.” Available: *http://imagic.weizmann.ac.il/~dov/freesw/FriBidi/.* Retrieved: July 17, 2000.

[35] Hall, William S. “Internationalization in Windows NT, Part:1 Programming with Unicode.” *Microsoft Systems Journal,* June 1994, pp 57-71.

[36] Holmes, Neville. “Toward Decent Text Encoding.” *IEEE Computer,* 1998, Number 8, Volume 31, August, pp 108-109.

[37] Homes, Nigel. “An Introduction to Pictoral Symbols.” in *Designing Pictoral Symbols,* Watson-Guptill, 1990.

[38] Horton, William. *The Icon Book: Visual Symbols for Computer Systems and Documentation.* John Wiley & Sons. 1994.

[39] Hughes, John. “Why Functional Programming Matters.” *Computer Journal,* 1989, Volume 32, Number 2, pp 98-107.

[40] International Business Machines Corporation. *National Language Design Guide, NLDG Volume 2.* IBM Canada Ltd. 1994.

[41] International Business Machines Corporation. *National Language Support Bidi Guide, NLDG Volume 3.* IBM Canada Ltd. 1995.

[42] International Business Machines Corporation. *MBCS/DBCS Character Set and Code Page System Architecture.* IBM Japan Ltd. 1996.

[43] International Business Machines Corporation. *MBCS Cross System Guide: Volume1 - Character Set and Code Page.* IBM Japan Ltd. 1997.

[44] International Business Machines Corporation. *OS/2 Warp Server for e-business Keyboards and Code Pages.* IBM. 1999.

[45] International Business Machines Corporation. “IBM Classes for Unicode.” Available: *http://www.ibm.com/java/tools/international-classes/index.html.* Retrieved: July 17, 2000.

[46] International Organization for Standardization. “ISO 7-bit Coded Character Set for Information Interchange.” *International Standard ISO/IEC 646:1991,* 1991.

[47] International Organization for Standardization. “8-bit Single-Byte Coded Graphic Character Sets - Part 1: Latin Alphabet No. 1.” *International Standard ISO/IEC 8859-1: 1998,* 1998.

[48] Ishida, Richard. “Non-Latin Writing Systems: Characteristics and Impact on Multinational Product Design.” *18th International Unicode Conference,* April 2001.

[49] Jennings, Tom. “ASCII: American Standard Code for Information Infiltration.” Available: *http://fido.wps.com/texts/codes.* Retrieved: January 9, 2001.

[50] Jones, Scott. et. al. *Digital Guide to Developing International User Information.* Digital Press. 1992.

[51] Jones, Simon P. et al. “Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language.” *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106,* February 1999.

[52] Kano, Nadine. *Developing International Software For Windows 95 and Windows NT.* Microsoft Press. 1995.

[53] Kataoka, Tomoko I. et al. “Internationalized Text Manipulation Covering Perso-Arabic Enhanced for Mongolian Scripts.” *Lecture Notes in Computer Science,* 1998, Volume 1375, pp 305-318.

[54] Korpela, Jukka. "A Tutorial on Character Code Issues." Available: *http://www.hut.fi/u/jkorpela/chars.html.* Retrieved: January 9, 2001.

[55] Lehtola, Aarno and Honkela, Timo. "A Framework for Global Software." *Proceedings of the 1st ERCIM Workshop on 'User Interfaces for All',* 1995.

[56] Leisher, Mark. "The UCData Unicode Character Properties and Bidi Algorithm Package." Available: *http://crl.nmsu.edu/~mleisher/ucdata.html.* Retrieved: July 17, 2000.

[57] Lunde, Ken. *CJKV Information Processing.* O'Reilly. 1999.

[58] Lunde, Ken. "A New Standard For Japanese." *Multilingual Computing and Technology,* 2000, Number 35, Volume 11, Issue 7, pp 45-46.

[59] Lunde, Ken. "CJKV Character Set and Encoding Developments." *Multilingual Computing and Technology,* 2001, Number 39, Volume 12, Issue 3, pp 53-55.

[60] Lunde, Ken "What's New In Unicode 3.1." *Multilingual Computing and Technology,* 2001, Number 42, Volume 12, Issue 6, p 51.

[61] Luong, Tuoc V. et. al. *Internationalization Developing Software for Global Markets*. John Wiley and Sons Inc. 1995.

[62] MacKay, Pierre. "Typesetting Problem Scripts." *Byte Magazine,* 1986, Volume 11, Number 2, pp 201-218.

[63] Madell, Tom. et. al. *Developing and Localizing International Software.* Prentice Hall. 1994.

[64] Maeda, Akira. "Studies on Multilingual Information Processing." *Doctor's Thesis Nara Institute of Science and Technology,* September 18, 2000.

[65] Math Forum, The. "Math Typesetting for the Internet." Available: *http://forum.swarthmore.edu/typesetting/.* Retrieved: August 13, 2001.

[66] Meyer, Dirk. “New Hong Kong Character Standard.” *Multilingual Computing and Technology,* 2000, Number 30, Volume 11, Issue 2, pp 30-32.

[67] Meyer, Dirk. “A New Chinese Character Set Standard.” *Multilingual Computing and Technology,* 2001, Number 37, Volume 12, Issue 1, pp 63-68.

[68] Meyer, Dirk. “Two New Chinese Character Standards: HK SCS & GB 18030-2000.” *18th International Unicode Conference,* April 2001.

[69] Microsoft. “TrueType Open Font Specification.” version 1.0. July 1995.

[70] Miller, Gary. *Personal Correspondence.* September 21, 2001.

[71] Milo, Thomas. “Creating Solutions for Arabic: A Case Study.” *18th International Unicode Conference,* April 2001.

[72] Morrison, Michael. et al. *XML Unleashed*. Sams Publishing. 1999.

[73] Mount Tahoma High School. “Japanese Tutorial.” Available: *http://www.tacoma.k12.wa.us/schools/hs/mount_tahoma/dept/japanese/.* Retrieved: August 24, 2001.

[74] Mudawwar, Muhammad F. “Multicode: A Truly Multilingual Approach to Text Encoding.” *IEEE Computer,* 1997, Number 4, Volume 30, April, pp 37-43.

[75] O’Donnell, Sandra M. *Programming for the World - A Guide to Internationalization.* Prentice Hall. 1994.

[76] Ohta, Masataka. “On Plain Text.” International Symposium on Multilingual Information Processing, Tsukuba Japan, March 25-26, 1996.

[77] Omniglot. “Phonetic Transcription of Chinese.” Available: *http:www.omniglot.com/writing/chinese2.htm.* Retrieved: August 30, 2001.

[78] Osawa, Noritaka. “EPICS: An Efficient, Programmable and Interchangeable Code System for WWW.” *6th International World Wide Web Conference*, April, 1997.

[79] Osawa, Noritaka. “A Multilingual Information Processing Infrastructure for Global Digital Libraries: EPICIST.” *Proceedings of the International Symposium on Research, Development and Practice in Digital Libraries,* November, 1997.

[80] Rossini, Stephane. *Egyptian Hieroglyphics.* Dover Publications. 1989.

[81] Ruesch, Jurgen and Kees, Weldon. “The Language of Identification and Recognition.” in *Nonverbal Communication,* Berkeley: University of California Press. 1970.

[82] Sakamura, Ken. “The TAD Language Environment and Multilingual Handling.” *TRONWARE*, 1992, Volume 50, pp 49-57.

[83] Salomon, Gitta. “New Uses for Color.” in *The Art of Human Computer Interface Design.* edited by Brenda Laurel, Addison-Wesley. 1990.

[84] Scherer, Markus. “GB 18030: A Mega-Codepage.” Available: *http:www-106.ibm.com/developerworks/unicode/library/u-china.html.* Retrieved: August 31, 2001.

[85] Schmitt, David A. *International Programming for Microsoft Windows*. Microsoft Press. 2000.

[86] Searfoss, Glenn. *JIS-Kanji Character Recognition.* Van Nostrand Reinhold. 1994.

[87] Smura, Edwin J. and Provan, Archie D. “Toward a New Beginning: The Development of a Standard for Font and Character Encoding to Control Electronic Document Interchange.” *IEEE Transactions on Professional Communication.* 1987, Number 4, Volume PC-30, pp 259-264.

[88] Stallman, Richard. *GNU Emacs Manual.* Free Software Foundation. 1999.

[89] Sun Microsystems. “Complex Text Layout Language Support in the Solaris Operating Environment.” Available: *http://www.sun.com/software/white-papers/wp-cttlanguage/.* Retrieved: July 17, 2000.

[90] Tanenbaum, Andrew S. *Computer Networks.* Prentice Hall. 1996.

[91] Taylor, David. *Global Software: Developing Applications for the International Market.* Springer-Verlag. 1992.

[92] Turley, James. “Computing In Vietnamese.” *Multilingual Computing and Technology,* 1998, Number 20, Volume 9, Issue 4, pp 25-29.

[93] Turley, James. “Computing In Chinese Poses Unique Challenges.” *Multilingual Computing and Technology,* 1999, Number 27, Volume 10, Issue 5, pp 30-33.

[94] Turley, James. “Computing in Chinese.” *Multilingual Computing and Technology,* 2000, Number 28, Volume 10, Issue 6, pp 28-30.

[95] Tuthill, Bill. and Smallberg, David. *Creating Worldwide Software.* Sun Microsystems Press. 1997.

[96] Unicode Consortium, The. *The Unicode Standard, Version 3.0.* Addison-Wesley. 2000.

[97] Unicode Consortium, The. “Unicode 3.0.1.” Available: *http://www.unicode.org/unicode/standard/versions/Unicode3.0.1.html.* Retrieved: January 17, 2001.

[98] Unicode Consortium, The. “Unicode 3.1.” Available: *http://www.unicode.org/unicode/standard/versions/Unicode3.1.html.* Retrieved: August 13, 2001.

[99] Unicode Consortium, The. “Plane 14 Characters for Language Tags.” Available: *http://www.unicode.org/reports/tr7.* Retrieved: January 9, 2001.

[100] Unicode Consortium, The. "Unicode Technical Report #9 - The Bidirectional Algorithm." Available: *http://www.unicode.org/unicode/reports/tr9/tr9-6.html* Retrieved: July 17, 2000.

[101] Unicode Consortium, The. "Unicode Standard Annex #15 - Unicode Normalization Forms." Available: *http://www.unicode.org/unicode/reports/tr15.* Retrieved: January 9, 2001.

[102] Van Camp, David. "Unicode and Software Globalization." *Dr. Dobb's Journal,* 1994, March, pp 46-50.

[103] Vine, Andrea. "An Overview Of The Unicode Standard 2.1." *Multilingual Computing and Technology,* 1998, Number 23, Volume 10, Issue 1, pp 50-52.

[104] Vine, Andrea. "Demystifying Character Sets." *Multilingual Computing and Technology,* 1999, Number 26, Volume 10, Issue 4, pp 48-52.

[105] Walters, Richard F. "Design of a Bitmapped Multilingual Workstation." *IEEE Computer,* 1990, Number 2, Volume 23, February, pp 33-41.

[106] Weider, Chris. et. al. "The Report of the IAB Character Set Workshop." RFC 2130, April 1997.

[107] World Wide Web Consortium, The. "Character Model for the World Wide Web." Available: *http://www.w3.org/TR/charmod*. Retrieved April 10, 2001.

[108] X/Open. *X/Open Internationalisation Guide.* X/Open Company Ltd. 1992.

[109] Yau, Michael M. T. "Supporting the Chinese, Japanese, and Korean Languages in the OpenVMS Operating System." *Digital Technical Journal,* 1993, Number 3, Volume 5, pp 63-79.

[110] Yergeau, F. "UTF-8 A Transformation Format of Unicode and ISO-10646." RFC 2044, October 1996.

[111] Yevgrashina, Lada. “Azerbaijan Drops Cyrillic for Latin Script.” *Reuters*, August 3, 2001.

Appendix A

```
1    -- Rule P2, P3 determine base level of text from the first strong
2    -- directional character
3    p2_3 :: [Attributed] -> Int
4    p2_3 [] = 0
5    p2_3 ((_,L):xs) = 0
6    p2_3 ((_,AL):xs) = 1
7    p2_3 ((_,R):xs) = 1
8    p2_3 (_:xs) = p2_3(xs)
9
10   -- Rules X2 - X9
11   x2_9 :: [Int] -> [Bidi] -> [Bidi] -> [Attributed] -> [Level]
12   x2_9 _ _ _ [] = []
13   x2_9 (l:ls) os es ((x,RLE):xs)
14      = x2_9 ((add l R):l:ls) (N:os) (RLE:es) xs
15   x2_9 (l:ls) os es ((x,LRE):xs)
16      = x2_9 ((add l L):l:ls) (N:os) (LRE:es) xs
17   x2_9 (l:ls) os es ((x,RLO):xs)
18      = x2_9 ((add l R):l:ls) (R:os) (RLO:es) xs
19   x2_9 (l:ls) os es ((x,LRO):xs)
20      = x2_9 ((add l L):l:ls) (L:os) (LRO:es) xs
21   x2_9 ls os (e:es) ((x,PDF):xs)
22      | elem e [RLE,LRE,RLO,LRO] = x2_9 (tail ls) (tail os) es xs
23   x2_9 ls os es ((x,PDF):xs)
24      = x2_9 ls os es xs
25   x2_9 ls os es ((x,y):xs)
26      | (head os) == N = ((head ls),x,y) : x2_9 ls os es xs
27      | otherwise = ((head ls),x,(head os)) : x2_9 ls os es xs
28
29   -- Rule X10 group characters by level
30   x10 :: (Int, Int) -> [Level] -> Run
31   x10 (sor,eor) xs
32      | even sor && even eor = LL xs
33      | even sor && odd eor = LR xs
34      | odd sor && even eor = RL xs
35      | otherwise = RR xs
36
37   -- Process explicit characters X1 - X10
38   explicit :: Int -> [Attributed] -> [Run]
39   explicit l xs = zipWith x10 (runList levels l l) groups
40      where levels = (map (\x -> level (head x)) groups)
41      groups = groupBy levelEql (x2_9 [l][N][] xs)
42
43
44
45
```

```
46   -- Rules W1 - W7
47   w1_7 :: [Level] -> Bidi -> Bidi -> [Level]
48   w1_7 [] _ _ = []
49   w1_7 ((x,y,L):xs) _ _ = (x,y,L):(w1_7 xs L L)
50   w1_7 ((x,y,R):xs) _ _ = (x,y,R):(w1_7 xs R R)
51   w1_7 ((x,y,AL):xs) _ _ = (x,y,R):(w1_7 xs AL R)
52   w1_7 ((x,y,AN):xs) dir _ = (x,y,AN):(w1_7 xs dir AN)
53   w1_7 ((x,y,EN):xs) AL _ = (x,y,AN):(w1_7 xs AL AN)
54   w1_7 ((x,y,EN):xs) L _ = (x,y,L):(w1_7 xs L EN)
55   w1_7 ((x,y,EN):xs) dir _ = (x,y,EN):(w1_7 xs dir EN)
56   w1_7 ((x,y,NSM):xs) L N   = (x,y,L):(w1_7 xs L L)
57   w1_7 ((x,y,NSM):xs) R N   = (x,y,R):(w1_7 xs R R)
58   w1_7 ((x,y,NSM):xs) dir last = (x,y,last):(w1_7 xs dir last)
59   w1_7 ((a,b,ES):(x,y,EN):xs) dir EN =
60      (a,b,EN):(x,y,EN):(w1_7 xs dir EN)
61   w1_7 ((a,b,CS):(x,y,EN):xs) dir EN =
62      (a,b,EN):(x,y,EN):(w1_7 xs dir EN)
63   w1_7 ((a,b,CS):(x,y,EN):xs) AL AN  =
64      (a,b,AN):(x,y,AN):(w1_7 xs AL AN)
65   w1_7 ((a,b,CS):(x,y,AN):xs) dir AN  =
66      (a,b,AN):(x,y,AN):(w1_7 xs dir AN)
67   w1_7 ((x,y,ET):xs) dir EN = (x,y,EN):(w1_7 xs dir EN)
68   w1_7 ((x,y,z):xs) dir last
69      | z==ET && findEnd xs ET == EN && dir /= AL
70         = (x,y,EN):(w1_7 xs dir EN)
71      | elem z [CS,ES,ET] = (x,y,ON):(w1_7 xs dir ON)
72      | otherwise = (x,y,z):(w1_7 xs dir z)
73
74   -- Process a run of weak characters W1 - W7
75   weak :: Run -> Run
76   weak (LL xs) = LL (w1_7 xs L N)
77   weak (LR xs) = LR (w1_7 xs L N)
78   weak (RL xs) = RL (w1_7 xs R N)
79   weak (RR xs) = RR (w1_7 xs R N)
80
81   -- Rules N1 - N2
82   n1_2 :: [[Level]] -> Bidi -> Bidi -> Bidi -> [Level]
83   n1_2 [] _ _ base = []
84   n1_2 (x:xs) sor eor base
85      | isLeft x = x ++ (n1_2 xs L eor base)
86      | isRight x  = x ++ (n1_2 xs R eor base)
87      | isNeutral x && sor == R && (dir xs eor) == R
88         = (map (newBidi R) x) ++ (n1_2 xs R eor base)
89      | isNeutral x && sor == L && (dir xs eor) == L
90         = (map (newBidi L) x) ++ (n1_2 xs L eor base)
91      | isNeutral x =
92         (map (newBidi base) x) ++ (n1_2 xs sor eor base)
93      | otherwise = x ++ (n1_2 xs sor eor base)
94
```

```
95  -- Process a run of neutral characters N1 - N2
96  neutral :: Run -> Run
97  neutral (LL xs) = LL (n1_2 (groupBy neutralEql xs) L L L)
98  neutral (LR xs) = LR (n1_2 (groupBy neutralEql xs) L R L)
99  neutral (RL xs) = RL (n1_2 (groupBy neutralEql xs) R L R)
100 neutral (RR xs) = RR (n1_2 (groupBy neutralEql xs) R R R)
101
102
103 -- Rule I1, I2
104 i1_2 :: [[Level]] -> Bidi -> [Level]
105 i1_2 [] _ = []
106 i1_2 ((x:xs):ys) dir
107   | attrib x == R && dir == L
108     = (map (newLevel 1) (x:xs)) ++ (i1_2 ys L)
109   | elem (attrib x) [AN,EN] && dir == L
110     = (map (newLevel 2) (x:xs)) ++ (i1_2 ys L)
111   | elem (attrib x) [L,AN,EN] && dir == R
112     = (map (newLevel 1) (x:xs)) ++ (i1_2 ys R)
113 i1_2 (x:xs) dir = x ++ (i1_2 xs dir)
114
115 -- Process a run of implicit characters I1 - I2
116 implicit :: Run -> Run
117 implicit (LL xs) = LL (i1_2 (groupBy bidiEql xs) L)
118 implicit (LR xs) = LR (i1_2 (groupBy bidiEql xs) L)
119 implicit (RL xs) = RL (i1_2 (groupBy bidiEql xs) R)
120 implicit (RR xs) = RR (i1_2 (groupBy bidiEql xs) R)
121
122 -- If a run is odd (L) then reverse the characters
123 reverseRun :: [Level] -> [Level]
124 reverseRun [] = []
125 reverseRun (x:xs)
126   | even (level x) = x:xs
127   | otherwise = reverse (x:xs)
128
129 reverseLevels :: [[Level]] -> [[Level]] -> Int -> [[Level]]
130 reverseLevels w [] _ = w
131 reverseLevels w (x:xs) a = if (level (head x)) >= a
132   then reverseLevels (x:w) xs a
133   else w ++ [x] ++ (reverseLevels [] xs a)
134
135 -- Rule L2 Reorder
136 reorder:: [Run] -> Bidi -> [[Level]]
137 reorder xs base = foldl (reverseLevels []) runs levels
138   where
139   flat = concat (map toLevel xs)
140   runs = map reverseRun (groupBy levelEql flat)
141   levels = getLevels runs
142
143
```

```
144  -- Rule L4 Mirrors
145  mirror:: [Level] -> [Level]
146  mirror [] = []
147  mirror ((x,y,R):xs) = case getMirror y of
148                  Nothing -> (x,y,R):(mirror xs)
149                  Just a -> (x,a,R):(mirror xs)
150  mirror (x:xs) = x:(mirror xs)
151
152  logicalToDisplay :: [Attributed] -> [Utf-32]
153  logicalToDisplay attribs
154     =let baseLevel = p2_3 attribs in
155       let baseDir = (if odd baseLevel then R else L) in
156       let x = explicit baseLevel attribs in
157       let w = map weak x in
158       let n = map neutral w in
159       let i = map implicit n in
160       map character (mirror (concat (reorder i baseDir)))
```

Appendix B

```
1    -- Unicode metadata tags
2    dirL = map intToWord32 [0xe0044,0xe0049,0xe0052,0xe0002,0xe004c,0xe0001]
3    dirR = map intToWord32 [0xe0044,0xe004c,0xe0052,0xe0002,0xe0052,0xe0001]
4    dirEnd = map intToWord32 [0xe007f,0xe0044,0xe0049,0xe0052,0xe0001]
5    parL = map intToWord32 [0xe0050,0xe0041,0xe0052,0xe0002,0xe004c,0xe0001]
6    parR = map intToWord32 [0xe0050,0xe0041,0xe0052,0xe0002,0xe0052,0xe0001]
7    parEnd = map intToWord32 [0xe007f,0xe0050,0xe0041,0xe0052,0xe0001]
8
9    -- Mark the level with the bidi tags
10   tagLevel :: Int -> [Level] -> [Ucs4]
11   tagLevel _ [] = []
12   tagLevel level ((x,y,z):xs)
13     | level /= x && even x
14       = dirL ++ (map character ((x,y,z):xs)) ++ dirEnd
15     | level /= x && odd x
16       = dirR ++ (map character ((x,y,z):xs)) ++ dirEnd
17     | otherwise
18       = map character ((x,y,z):xs)
19
20   -- Mark the run with the bidi tags
21   tagRun :: Int -> Run -> [Ucs4]
22   tagRun z (LL xs) = parL ++ concat (map (tagLevel z)
23     (groupBy levelEql (mirror xs))) ++ parEnd
24   tagRun z (LR xs) = parL ++ concat (map (tagLevel z)
25     (groupBy levelEql (mirror xs))) ++ parEnd
26   tagRun z (RL xs) = parR ++ concat (map (tagLevel z)
27     (groupBy levelEql (mirror xs))) ++ parEnd
28   tagRun z (RR xs) = parR ++ concat (map (tagLevel z)
29     (groupBy levelEql (mirror xs))) ++ parEnd
30
31   -- Insert mirror tags
32   mirror :: [Level] -> [Level]
33   mirror [] = []
34   mirror ((x,y,R):xs)
35     | isMirrored y
36       = (x,0xe004d,R):(x,0xe0049,R):(x,0xe0052,R):(x,y,R)
37         : mirror xs
38     | otherwise = (x,y,R) : (mirror xs)
39   mirror (x:xs) = x : (mirror xs)
```

Appendix C

```
1      import java.util.*;
2      import java.io.*;
3
4      public class UniMeta {
5          BufferedReader dataIn;
6          String dirL = "\udb40\udc44\udb40\udc49\udb40\udc52\udb40\udc02\udb40\udc4c" +
7                        "\udb40\udc01";
8         dirR = "\udb40\udc44\udb40\udc49\udb40\udc52\udb40\udc02\udb40\udc52" +
9                  "\udb40\udc01";
10        dirEnd = "\udb40\udc7f\udb40\udc44\udb40\udc49\udb40\udc52\udb40\udc01";
11        parL = “\udb40\udc50\udb40\udc41\udb40\udc52\udb40\udc02\udb40\udc4c” +
12                 “\udb40\udc01”;
13        parR = “\udb40\udc50\udb40\udc41\udb40\udc52\udb40\udc02\udb40\udc52” +
14                 “\udb40\udc01”;
15        parEnd = "\udb40\udc7f\udb40\udc50\udb40\udc41\udb40\udc52\udb40\udc01";
16        mirror = "\udb40\udc4d\udb40\udc49\udb40\udc52";
17
18        String lBDO = "<bdo dir=\"ltr\">",
19           rBDO = "<bdo dir=\"rtl\">",
20           lP = "<p dir=\”ltr\”>",
21           rP = “<p dir=\”rtl\”>”,
22           endP = "</p>",
23           endBDO ="</bdo>";
24         // Open the input file
25         public UniMeta(String in) {
26             try {
27                 FileInputStream fileIn = new FileInputStream(in);
28                 InputStreamReader str =
29                        new InputStreamReader(fileIn, "UTF8");
30                        dataIn = new BufferedReader(str);
31             }
32             catch (Exception e) {
33                 System.out.println("Error opening file " + in);
34                 return;
35             }
36         }
37         // Replace the unicode meta tags with HTML tags
38         private String replace(String in) {
39              StringBuffer out = new StringBuffer();
40              int i = 0;
41                  while(i < in.length()) {
42                      if (in.startsWith(parL, i)) {
43                           out.append(lP+lBDO);
44                           i += parL.length();
45                      }
```

```
46                  else if (in.startsWith(parR, i)) {
47                      out.append(rP+rBDO);
48                      i += parR.length();
49                  }
50               else if (in.startsWith(dirL, i)) {
51                  out.append(lBDO);
52                  i += dirL.length();
53               }
54               else if (in.startsWith(dirR, i)) {
55                  out.append(rBDO);
56                  i += dirR.length();
57               }
58               else if (in.startsWith(dirEnd, i)) {
59                  out.append(endBDO);
60                  i += dirEnd.length();
61               }
62                  else if (in.startsWith(parEnd, i)) {
63                  out.append(endBDO+endP);
64                  i += parEnd.length();
65               }
66               else if (in.startsWith(mirror, i)) {
67                  i += mirror.length();
68               }
69               else {
70                  out.append(in.charAt(i));
71                  ++i;
72               }
73            }
74            return (out.toString());
75        }
76
77        // Process the input stream, generate output to stdio
78        public void parse() {
79           String in = null;
80           System.out.println("<html>");
81           try {
82              while ((in = dataIn.readLine()) != null) {
83                 System.out.println(replace(in));
84              }
85           }
86           catch(Exception e) {
87              System.out.println("Error parsing file");
88              return;
89           }
90           System.out.println("</html>");
91        }
92
93
94
```

```
95    public static void main(String[] args) {
96            UniMeta input = new UniMeta(args[0]);
97            input.parse();
98        }
99    }
```

Appendix D

```
1    module Metacode where
2
3    import Unicode
4    import Word
5
6    type MetaChar = Ucs4
7
8    listFilter :: Eq a => (a -> Bool) -> [a] -> [a]
9    listFilter _ [] = []
10   listFilter f (x:xs) = if f x then listFilter f xs else x:(listFilter f xs)
11
12   listEqual :: Eq a => [a] -> [a] -> Bool
13   listEqual [] [] = True
14   listEqual [] _ = False
15   listEqual _ [] = False
16   listEqual (x:xs) (y:ys) = if x == y then listEqual xs ys else False
17
18   isMetadata :: MetaChar -> Bool
19   isMetadata x
20      | x >= 0xe0000 && x <= 0xe007f = True
21      | otherwise = False
22
23   byteEquivalent :: [Word8] -> [Word8] -> Bool
24   byteEquivalent xs ys = listEqual xs ys
25
26   codePointEquivalent :: [MetaChar] -> [MetaChar] -> Bool
27   codePointEquivalent xs ys = listEqual xs ys
28
29   contentEquivalent :: [MetaChar] -> [MetaChar] -> Bool
30   contentEquivalent xs ys
31      = let fxs = listFilter isMetadata xs
32            fys = listFilter isMetadata ys in
33            codePointEquivalent fxs fys
```