

The use of international encoding standards for local language computer software, documents and data.

*Authors: Marcus Wright, Computer Science Department, Lynchburg College,
William Briggs, Computer Science Department, Lynchburg College,
Robert Van Buskirk, Eritrea Technical Exchange Project/ICSEE
and Craig Harmer, Eritrea Technical Exchange Project/ICSEE*

ABSTRACT

Until recently Ge'ez computer documents in Eritrea utilized non-standard encodings for characters. Since 1998, international standards for the encoding of Ge'ez letters has been in existence. In this paper we describe the international computer encoding standards for Ge'ez letters and the recent development of software that allows computer users to create Ge'ez documents that are standard-compliant. We also discuss software property issues and the use of a public open-source license as a mechanism for developing intellectual software property. We conclude with a discussion of plans for future multi-lingual computer utility development plans.

Introduction

Making Eritreans more comfortable with computers and computer-based information will be a crucial element of bringing the benefits of the Internet and computers to the average Eritrean.

The purpose of this paper is to provide a detailed description, background, and reference for recent efforts to standardize multi-lingual computer use and software for Eritrea. During the past year we have created a standard compliant Ge'ez software and conversion utility called UniGeez that is freely available under an open source software license. This work has been sponsored by the Eritrean Technical Exchange, a project of the 501(c)(3) non-profit International Collaborative for Science Education and the Environment (ICSEE), and performed in collaboration with the Computer Science Department of Lynchburg College in Lynchburg, Virginia.

In this paper we first discuss some the motivation, demand, and need for standard-compliant multi-lingual computer capabilities in Eritrea. We follow with a discussion of the issues relevant to multi-lingual computer use. We then discuss multi-lingual encodings and encoding standards. We proceed to a presentation of the most popular Ge'ez transliteration standards and schemes in Eritrea. We then provide a detailed discussion of the standard-compliant UniGeez solution for multi-lingual computer needs in Eritrea. We follow with a discussion of some of the software property/ownership issues. And we conclude with an outline of plans and ideas regarding further development of multi-lingual computer information capacity in Eritrea and the Eritrean diaspora.

Motivation

Limited access to technology (and the enhanced productivity it brings) is one of the main barriers to raising the standard of living and the value of national economic production in Eritrea. The ease of computer access, and the relevance of computer information will be a major element of rapidly transferring and applying computer technologies to Eritrea's economic and productive activities.

The for-profit private sector development model for multi-lingual computer infrastructure has failed Eritrea. First of all there are very few multi-lingual software providers and developers in Eritrea (there are approximately two), and the software that has been developed is expensive, uses non-standard character encodings, and is mutually incompatible with other software in Eritrea and Ethiopia. Prices for existing software range from \$20 to \$90 per copy and there were no free versions of the software until UniGeez began free distribution. This had led to the contradiction that English-speaking people can use computers in their own language for free, while Tigrigna speakers (who have a mean per-capital of \$250/year) may have to pay \$20 to use a computer in their own language.

The large public benefits of multi-lingual computer access means that basic utilities that provide easy multi-lingual computer access should be public, rather than private property. A person trained and proficient in computers has an earning potential of perhaps 2-10 times that of a person without computer training. If free, public, multi-lingual computer utilities can facilitate enhanced computer access for just 1000 more people, then the national economic benefit can be as much as 1000 people*\$1000/yr = \$1 million/year. This justifies significant public sector investment in the development of basic multi-lingual computer infrastructure.

Demand for multi-lingual capabilities

Virtually all Eritreans who use computers have the need to produce documents in Tigrigna and/or Arabic. This market by now consists of tens of thousands of users who are probably producing hundreds of thousands of documents each year. Up until the year 2000, this market was met by primarily two multi-lingual software providers in Eritrea. Tfanus Enterprises (<http://www.tfanus.com.er/>) who markets a program called Yada that provided the capability of writing documents in both Ge'ez and Arabic, and Phonetic Systems (<http://www.geezsoft.com/>) which provides the software Geez Gateway for producing documents in Tigrigna. Both companies are rather small (less than 10 employees), and revenues from software sales (though not known in detail) are probably only a few tens of thousands of dollars per year.

But in spite of the rather limited software sales, demand for multi-lingual computer services and systems is growing at a rapid pace that reflects the rapid growth of the computer and technology sector in Eritrea.

Infrastructure needs for multi-lingual software

In Eritrea documents are produced and maintained in English, Tigrigna, and Arabic (along with other, less common languages). Most multi-lingual computer use consists of routine word processing and document production as part of organizational communication and administration activities. Another popular application of multi-lingual software is desktop publishing activities. As Eritrea's computer sector gets more developed, the need for database applications,

website development and maintenance, and multi-lingual communication and documentation services will grow. Eventually such multi-lingual computer applications will have to follow international norms and standards in order to be sustainable. As Eritrea's computer infrastructure develops, trained experts and support personnel will also have to provide a roughly consistent and compatible implementation of multi-lingual features for the sake of economic efficiency. Consistent, efficient information production and document exchange is a basic infrastructure requirement of Eritrea's computer systems and communications development.

Why Standard Compliance?

The efficiency and productivity of computer communications depends directly on the speed and cost of transferring information from one person or application to another. Currently, in the U.S. the largest amount of time is spent in getting information is not the network transfer, but locating and reading the information. Perhaps it takes 15 seconds to go to a search engine, a minute to find the page in the search engine results, and another minute to read the information. If documents are not prepared in a consistent encoding and format, then another step may have to be taken to read or use information that is in Ge'ez because of the need to convert or translate the character encoding between formats. That step, even if it takes less than a minute, can increase the time of presenting or retrieving information by up to 30%. In addition, developers and content providers would have to spend extra resources to provide translation and conversion facilities for different types of Ge'ez. Even worse, after standards become dominant, the cumulative archives of non-standard Ge'ez documents will have to be converted to be useful. A reasonable estimate of the costs of conversion and conversion support in a computer communications environment without standards is about 10% of computer communications activity.

Current computer communications markets in Eritrea are running at more than \$300,000 per year and doubling at about 100%/year. The computer services sector is probably 5 to ten times this amount. This means that the cost of non-compliance with standards can be tens to hundreds of thousands of dollars per year in the near future. Mostly this cost is reflected in the lost opportunities of people using English letters and text when they would be much happier and effective in using Ge'ez or Arabic if it was convenient and readily available.

Languages on the computer

Language elements: encoding, glyphs/fonts, and keyboard mapping/input method

The representation of different written languages on a computer has three elements: the encoding, the font, and the input method. An optional fourth element is a transliteration scheme.

The language *encoding* indicates how the different letters and symbols of a written language are represented in the computer's memory as binary numbers. The binary numbers in a computer are usually represented as bytes of data, where one byte is an 8-digit binary number that can range from 0 to 255. The encoding method specifies how the different letters are stored in one to several bytes of data.

The *font maps* a particular binary encoded character to an image. This means that the same encoded letters can have different fonts or 'glyphs' that represent different artistic and typographic styles of representing the characters of a language.

The *input method* specifies how different combinations of keystrokes from the keyboard will be translated into different characters. For people who use English or American keyboards, it is often convenient to map the English letters to letters of another language in a phonetic way that is easy to remember. A systematic mapping from one language's characters to another language's characters is called a transliteration scheme.

To write Ge'ez on a computer with Ge'ez characters all three elements are necessary: an encoding, a font, and an input method. Until recently in Eritrea, all three elements would be developed independently by different developers of multi-lingual software. But in the last several years international standards have been developed for the encoding scheme, while a diversity of unrelated developers are starting to develop different fonts based on the international encoding standard.

Encodings for Ge'ez

History and Background of Encoding Standards

When computers were initially developed in the U.S., the main concern was to encode the various letters of the English alphabet and associated symbols as binary numbers. The initial standard on encoding letters as digital numbers was referred to as ASCII: The American Standard Code for Information Interchange, was proposed by the American National Standards Institute in 1963, and adopted as ANSI X3.4 in 1968 [Czyborra, 1998]. ASCII is a seven-bit (128 character) encoding standard. In 1972 ASCII and different non-English variants were incorporated as a more general standard that allowed encoding for about 14 languages as ISO 646, and this was later extended as a more general 8-bit western language encoding standard ISO-8859-1.

But the western encoding standard was obviously inadequate for many of the world's non-western languages, and a completely universal international encoding standard was developed:

"The even wider Universal Character Set (UCS, Unicode) wants to simplify world-wide multilingual and multi-platform text exchange just like the ASCII standard simplified Latin text interchange and aims to become equally successful as the ultimate encoding standard and thus it was numbered ISO 10646 = ISO 646 + 10000. And the inevitable success seems to be there: RFC 2070 internationalized the Internet's hypertext markup language HTML and declared ISO 10646 its new base charset. RFC 2277 recommends the use of ISO 10646 to all new Internet protocols.

"Unicode begins with US-ASCII and ISO-8859-1 but goes beyond the 8bit barrier and encodes all the world's characters in a 16bit space and a 20bit extension zone for everything that did not fit into the 16bit space. The ASCII-compatible Unicode transformation scheme UTF-8 lets all ASCII characters pass through transparently and encodes all other characters as unambiguous 8bit character sequences." [Czyborra, 1998]

The Unicode standard provides for a "Universal Multiple-Octet Coded Character Set (UCS)" which--because it is using several bytes of data to encode the characters--can include characters from essentially all languages. The Unicode standards have been published and updated by the International Standards Organization (ISO). The Unicode standard has gone through several versions and the Ge'ez characters were incorporated in October of 1998 as ISO 10646-1 amendment 10.

The advantage of multi-byte characters is that they are large enough to include all characters from almost all languages. One problem is how to make such a scheme compatible with the older ASCII standard, and how to represent them in a way that makes it clear when one character ends and the next begins (since now the characters have a variable length). This second problem is solved with the UTF-8 encoding scheme which is part of the Unicode standard.

UTF-8

What the UTF-8 encoding does is provide a means of encoding both ASCII and Unicode characters using a representation of several 8-bit characters. The way it does this is by reserving the first bit of the 8-bit character as an unicode/ascii character indicator or switch. If the first bit is a 0, the character is a 7-bit ASCII character, while if the first bit is 1, the character will be interpreted as part of a unicode character. And if the first bit is 1, then the second bit indicates whether or not the byte is the first byte of a multi-byte UTF-8 unicode character, or if the character is holding data for a multi-byte character. Essentially, in the UTF-8 standard, each 8-bit character acts as a data packet, and the initial bits of the character act as flags for the type of binary data stored, and how it is organized over several bytes. The following table is modified from the Master's thesis of Roman Czyborra [Czyborra, 1998] and indicates the lead bit patterns, and the data bit locations for character data storage under the UTF-8 standard. Under this standard character of up to 31 bits is possible, though for Ge'ez, only 16 bits are used:

bytes	bits	representation
1	7	0xxxxxxx
2	11	110xxxxx 10xxxxxx
3	16	1110xxxx 10xxxxxx 10xxxxxx
4	21	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
5	26	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
6	31	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

For Ge'ez, the unicode encoding of the binary character data is two-bytes or 16 bits per character, so in UTF-8 the character data is stored in three 8-bit bytes where the first four bits of the first byte are 1110, and the first two bits of the subsequent bytes are 10.

Computer Application Support for UTF-8 and Unicode

Since the Unicode and UTF-8 encoding schemes are extensions of the old ASCII and Western character encoding schemes, older applications and operating systems that were developed only for ASCII and Western characters may not support the Unicode standard and characters. This is one of the main barriers to using standard compliant Ge'ez software at this time in Eritrea and in the Eritrean community. But even though it is currently a barrier, over time more and more software, applications and operating systems will be compliant with the Unicode standard (ISO 10646-1). Currently almost all software and operating systems with global distribution and with an age less than a few years old is ISO-10646-1 standard-compliant.

Ge'ez Encodings in Eritrea

We are hopeful that within a few years there will be one major Ge'ez encoding scheme in Eritrea that complies--as much as possible--with the international ISO-10646-1 standard. But between the present and the time of standardized encodings, there will have to be a transition period where people convert from pre-Unicode encodings to the Unicode standardized encodings.

Prior to the Unicode standard, Ge'ez program developers in Ethiopia and Eritrea had to come up with their own alternatives to encoding Ge'ez characters. The majority of these developers simply changed the fonts that they used for the Western ISO-8859-1 character set to replace the European characters with Ge'ez characters. They then wrote programs that would map the keyboard strokes to their particular character-based encoding, and the Ge'ez letters would be displayed by choosing the appropriate font.

There were two major problems with the pre-Unicode scheme. The first, and perhaps largest problem is that each developer has his own unique encoding scheme. As a result, over 70 different encoding schemes were developed for Eritrea and Ethiopia.

To try to overcome the confusion represented by the extreme diversity of encoding schemes, the LibETH project attempted to produce nearly universal text conversion software that would work with most of the different proprietary Ge'ez encodings [Yacob, 2001]. But the LibETH project to date has not had the resources to make user-friendly versions of this utility that could find wide application in the Eritrean community and diaspora. While we have benefitted greatly from the technical work done by the LibETH project, we hope that some of the need for the LibETH project also has been partially supplanted by the development of standard-compliant Ge'ez software. We have found development of accurate user-friendly software that converts between just three encodings to require approximately six months of work to perfect, and hope that we can avoid the amount of work that would be required to develop user-friendly software for 70 different encoding systems.

The two Ge'ez software systems that found wide use in Eritrea were *GeezGate* by Phonetic Systems and *Yada* by Tfanus enterprises. Both systems use non-standard fonts for Western ISO-8859-1 characters to display Ge'ez. But since there are less than 255 characters in ISO-8859-1 and about 318 characters and punctuation marks (more or less depending on the inclusion of less common characters), both systems break characters into a base character and re-usable character markings in order to fit all of the necessary pieces of characters into less than 255 slots. This results in variable-length and sometimes multi-byte character encodings for the Ge'ez characters. We provide tables of these variable-length multi-byte character encodings in an appendix.

Keyboard mappings, Transliteration Schemes and Input methods

There are four transliteration schemes or keyboard mappings for Ge'ez that are relevant to Eritrean applications. They consist of the "System for Ethiopic Representation in ASCII" (SERA), the Dehai g'z standard, the keyboard mapping for Phonetic Systems' Geez Gateway, and the keyboard mapping for Tfanus Enterprise's Yada.

Both SERA and the Dehai g'z standard are transliteration schemes. A transliteration scheme is a common-sense method for writing a language with a non-latin script in latin characters in a way that is consistent, unambiguous, and roughly phonetic.

SERA

The SERA transliteration system is best described in its original on-line documentation:

``SERA'' is an acronym for ``System for Ethiopic Representation in ASCII''. Most simply put -SERA is a way to write in Fidel script using Latin letters. More extendedly; SERA is a convention for transliteration of Fidel script into Latin that insures the integrity of the format and content of the original document, and that it be fully transportable across all computer mediums.

We have initially used the SERA transliteration scheme for our standard-compliant software because it was the best documented, complete scheme relevant to the maximum number of characters for both Eritrean and Ethiopian languages. In the next section we also provide an extension of the Dehan g'z transliteration standard that may also be incorporated into our multi-lingual software in the future.

We present the transliteration standard in the form of a table with 41 rows and 12 columns. Each row represents a different class of letters, and each column represents a different form (e.g. he, hu, hi, ha, hie, h, ho), we also include columns 8, 9, 10, 11, and 12 for the labiovelar forms that end with the We, Wu, Wi, Wa, and Wie vowel sounds. We also include notes in parentheses for the language in which the characters are used if it is not a common Tigrigna character.

	1	2	3	4	5	6	7	8	9	10	11	12
	g`Iz	ka`Ib	sals	rab`I	hams	sads	sab`I	digala -->				
1	he	hu	hi	ha	hE	h	ho					
2	le	lu	li	la	lE	l	lo	lW/lWa				
3	He	Hu	Hi	Ha	HE	H	Ho	HW/HWa				
4	me	mu	mi	ma	mE	m	mo	mW/mWa				
5	`se	`su	`si	`sa	`sE	`s	`so	`sW/`sWa				
6	re	ru	ri	ra	rE	r	ro	rW/rWa				
7	se	su	si	sa	sE	s	so	sW/sWa				
8	xe	xu	xi	xa	xE	x	xo	xW/xWa				
9	qe	qu	qi	qa	qE	q	qo	qWe	qW/qWu	qWi	qWa	qWE
10	`qe	`qu	`qi	`qa	`qE	`q	`qo		(`q is Chaha)			
11	Qe	Qu	Qi	Qa	QE	Q	Qo	QWe	QW/QWu	QWi	QWa	QWE
12	be	bu	bi	ba	bE	b	bo	bW/bWa				
13	ve	vu	vi	va	vE	v	vo	vW/vWa				
14	te	tu	ti	ta	tE	t	to	tW/tWa				
15	ce	cu	ci	ca	cE	c	co	cW/cWa				
16	`he	`hu	`hi	`ha	`hE	`h	`ho	hWe	hW/hWu	hWi	hWa	hWE
17	ne	nu	ni	na	nE	n	no	nW/nWa				
18	Ne	Nu	Ni	Na	NE	N	No	NW/NWa				
19	e	u	i	a	E	I	o	ea				
20	ke	ku	ki	ka	kE	k	ko	kWe	kW/kWu	kWi	kWa	kWE
21	`ke	`ku	`ki	`ka	`kE	`k	`ko		(`k is Chaha)			
22	Ke	Ku	Ki	Ka	KE	K	Ko	KWe	KW/KWu	KWi	KWa	KWE
23	Xe	Xu	Xi	Xa	XE	X	Xo		(X is Chaha)			
24	we	wu	wi	wa	wE	w	wo					
25	`e	`u	`i	`a	`E	`I	`o					
26	ze	zu	zi	za	zE	z	zo	zW/zWa				
27	Ze	Zu	Zi	Za	ZE	Z	Zo	ZW/ZWa				
28	ye	yu	yi	ya	yE	y	yo	yW/yWa				
29	de	du	di	da	dE	d	do	dW/dWa				
30	De	Du	Di	Da	DE	D	Do	DW/DWa	(D is Oromiffa)			
31	je	ju	ji	ja	jE	j	jo	jW/jWa				
32	ge	gu	gi	ga	gE	g	go	gWe	gW/gWu	gWi	gWa	gWE
33	`ge	`gu	`gi	`ga	`gE	`g	`go		(`g is Chaha)			
34	Ge	Gu	Gi	Ga	GE	G	Go	GWe	GW/GWu	GWi	GWa	GWE
35	Te	Tu	Ti	Ta	TE	T	To	TW/TWa	(G is Bilin)			
36	Ce	Cu	Ci	Ca	CE	C	Co	CW/CWa				
37	Pe	Pu	Pi	Pa	PE	P	Po	PW/PWa				
38	Se	Su	Si	Sa	SE	S	So	SW/SWa				
39	`se	`su	`si	`sa	`sE	`s	`so					
40	fe	fu	fi	fa	fE	f	fo	fW/fWa				
41	pe	pu	pi	pa	pE	p	po	pW/pWa				

Source: http://www.abysiniacybergateway.net/fidel/sera-faq_0.html

Extended Dehai

One of the most popular transliteration standards amongst the Eritrean community is the Dehai transliteration standard. This standard chooses to use slightly

longer strings of characters to characterize the different letters of the Ge'ez syllabary. And it uses capitalization to characterize laryngeal or plosive forms of the consonants or sounds. We extend the dehai syllabary to include the SERA transliterations for the Bilin, Chaha, and Oromiffa characters.

	1	2	3	4	5	6	7	8	9	10	11	12
	gEz	kaEb	sals	rabE	hams	sads	sabE	diqala -->				
1	he	hu	hi	ha	hie	h	ho					
2	le	lu	li	la	lie	l	lo	lW/lWa				
3	He	Hu	Hi	Ha	Hie	H	Ho	HW/HWa				
4	me	mu	mi	ma	mie	m	mo	mW/mWa				
5	`se	`su	`si	`sa	`sie	`s	`so	`sW/`sWa				
6	re	ru	ri	ra	rie	r	ro	rW/rWa				
7	se	su	si	sa	sie	s	so	sW/sWa				
8	she	shu	shi	sha	shie	sh	sho	shW/shWa				
9	qe	qu	qi	qa	qie	q	qo	qWe	qW/qWu	qWi	qWa	qWie
10	`qe	`qu	`qi	`qa	`qie	`q	`qo					
11	Qe	Qu	Qi	Qa	Qie	Q	Qo	QWe	QW/QWu	QWi	QWa	QWie
12	be	bu	bi	ba	bie	b	bo	bW/bWa				
13	ve	vu	vi	va	vie	v	vo	vW/vWa				
14	te	tu	ti	ta	tie	t	to	tW/tWa				
15	che	chu	chi	cha	chie	ch	cho	chW/chWa				
16	`he	`hu	`hi	`ha	`hie	`h	`ho	hW/hWu	hWi	hWa	hWie	
17	ne	nu	ni	na	nie	n	no	nW/nWa				
18	Ne	Nu	Ni	Na	Nie	N	No	NW/NWa				
19	'e	'u	'i	'a	'ie	'	'o	'ea				
20	ke	ku	ki	ka	kie	k	ko	kWe	kW/kWu	kWi	kWa	kWie
21	`ke	`ku	`ki	`ka	`kie	`k	`ko					
22	Ke	Ku	Ki	Ka	Kie	K	Ko	KWe	KW/KWu	KWi	KWa	KWie
23	She	Shu	Shi	Sha	Shie	Sh	Sho					
24	we	wu	wi	wa	wie	w	wo					
25	Ae	U	I	A	Aie	E	O					
26	ze	zu	zi	za	zie	z	zo	zW/zWa				
27	Ze	Zu	Zi	Za	Zie	Z	Zo	ZW/ZWa				
28	ye	yu	yi	ya	yie	y	yo	yW/yWa				
29	de	du	di	da	die	d	do	dW/dWa				
30	De	Du	Di	Da	Die	D	Do	DW/DWa				
31	je	ju	ji	ja	jie	j	jo	jW/jWa				
32	ge	gu	gi	ga	gie	g	go	gWe	gW/gWu	gWi	gWa	gWie
33	`ge	`gu	`gi	`ga	`gie	`g	`go					
34	Ge	Gu	Gi	Ga	Gie	G	Go	GWe	GW/GWu	GWi	GWa	GWie
35	Te	Tu	Ti	Ta	Tie	T	To	TW/TWa				
36	Che	Chu	Chi	Cha	Chie	Ch	Cho	ChW/ChWa				
37	Pe	Pu	Pi	Pa	Pie	P	Po	PW/PWa				
38	Se	Su	Si	Sa	Sie	S	So	SW/SWa				
39	`Se	`Su	`Si	`Sa	`Sie	`S	`So					
40	fe	fu	fi	fa	fie	f	fo	fW/fWa				
41	pe	pu	pi	pa	pie	p	po	pW/pWa				

Source: <http://www.primenet.com/~ephrem2/languages/tigre/dehaistan.html>
with modifications by the authors

Tfanus

The Tfanus keyboard mapping is in some ways the most distinct of all of the schemes, with some of the logic driven by the ease of typing more than the phonetic connection to the Ge'ez syllables. Note for example that for the 'ie' sound the Tfanus/Yada keyboard uses a lower case 'v' rather than the upper case 'E' of SERA or the two-stroke 'ie' of the Dehai standard and Phonetic Systems.

	1	2	3	4	5	6	7	8	9	10	11	12
	gEz	kaEb	sals	rabE	hams	sads	sabE	diqala -->				
1	he	hu	hi	ha	hv	h	ho					
2	le	lu	li	la	lv	l	lo	lA				
3	He	Hu	Hi	Ha	Hv	H	Ho	HA				
4	me	mu	mi	ma	mv	m	mo	mA				
5	We	Wu	Wi	Wa	Wv	W	Wo	WA				
6	re	ru	ri	ra	rv	r	ro	rA				
7	se	su	si	sa	sv	s	so	sA				
8	Se	Su	Si	Sa	Sv	S	So	SA				
9	qe	qu	qi	qa	qv	q	qo	qO	qI	qI	qA	qV
10												
11	Qe	Qu	Qi	Qa	Qv	Q	Qo	QO	QI	QI	QA	QV
12	be	bu	bi	ba	bv	b	bo	bA				
13	Be	Bu	Bi	Ba	Bv	B	Bo	BA				
14	te	tu	ti	ta	tv	t	to	tA				

15	Ce	Cu	Ci	Ca	Cv	C	Co	CA				
16	Ue	Uu	Ui	Ua	Uv	U	Uo	hO	hI	hI	hA	hV
17	ne	nu	ni	na	nv	n	no	nA				
18	Ne	Nu	Ni	Na	Nv	N	No	NA				
19	Ee	Eu	Ei	Ea	Ev	E	Eo					
20	ke	ku	ki	ka	kv	k	ko	kO	kI	kI	kA	kV
21												
22	Ke	Ku	Ki	Ka	Kv	K	Ko	KO	KI	KI	KA	KV
23												
24	we	wu	wi	wa	wv	w	wo					
25	[e	[u	[i	[a	[v	[[o					
26	ze	zu	zi	za	zv	z	zo	zA				
27	Ze	Zu	Zi	Za	Zv	Z	Zo	ZA				
28	ye	yu	yi	ya	yv	y	yo	yA				
29	de	du	di	da	dv	d	do	dA				
30												
31	je	ju	ji	ja	jv	j	jo	jA				
32	ge	gu	gi	ga	gv	g	go	gO	gI	gI	gA	gV
33												
34												
35	Te	Tu	Ti	Ta	Tv	T	To	TA				
36	Ce	Cu	Ci	Ca	Cv	C	Co	CA				
37	Pe	Pu	Pi	Pa	Pv	P	Po	PA				
38	Se	Su	Si	Sa	Sv	S	So	SA				
39	Xe	Xu	Xi	Xa	Xv	X	Xo					
40	fe	fu	fi	fa	fv	f	fo	fA				
41	pe	pu	pi	pa	pv	p	po	pA				

Phonetic Systems

The phonetic systems keyboard or transliteration system provides mappings for only the characters in Tigrigna, Amharic, and Ge'ez. It is similar to the Deha method, except it uses 'ua' instead of Wa and an apostrophe after a vowel to for the various labiovelor forms. Also it uses brackets for some of the less common characters.

	1	2	3	4	5	6	7	8	9	10	11	12
	gOz	kaOb	sals	rabO	hams	sads	sabO	diqala -->				
1	he	hu	hi	ha	hie	h	ho					
2	le	lu	li	la	lie	l	lo	lua				
3	He	Hu	Hi	Ha	Hie	H	Ho					
4	me	mu	mi	ma	mie	m	mo	mua				
5	[e	[u	[i	[a	[ie	[[o	[ua				
6	re	ru	ri	ra	rie	r	ro	rua				
7	se	su	si	sa	sie	s	so	sua				
8	Se	Su	Si	Sa	Sie	S	So	Sua				
9	qe	qu	qi	qa	qie	q	qo	qe'	q'	qi'	qa'	qie'
10												
11	Qe	Qu	Qi	Qa	Qie	Q	Qo	Qe'	Q'	Qi'	Qa'	Qie'
12	be	bu	bi	ba	bie	b	bo	bua				
13	ve	vu	vi	va	vie	v	vo					
14	te	tu	ti	ta	tie	t	to	tua				
15	ce	cu	ci	ca	cie	c	co	cua				
16	le	lu	li	la	lie	l	lo	le'	l'	li'	la'	lie'
17	ne	nu	ni	na	nie	n	no	nua				
18	Ne	Nu	Ni	Na	Nie	N	No	Nua				
19	e	u	i	a	Aie	A	o					
20	ke	ku	ki	ka	kie	k	ko	ke'	k'	ki'	ka'	kie'
21												
22	Ke	Ku	Ki	Ka	Kie	K	Ko	Ke'	K'	Ki'	Ka'	Kie'
23												
24	we	wu	wi	wa	wie	w	wo					
25	Oe	Ou	Oi	Oa	Oie	O	Oo					
26	ze	zu	zi	za	zie	z	zo	zua				
27	Ze	Zu	Zi	Za	Zie	Z	Zo	Zua				
28	ye	yu	yi	ya	yie	y	yo					
29	de	du	di	da	die	d	do	dua				
30												
31	je	ju	ji	ja	jie	j	jo	jua				
32	ge	gu	gi	ga	gie	g	go	ge'	g'	gi'	ga'	gie'
33												
34												
35	Te	Tu	Ti	Ta	Tie	T	To	Tua				
36	Ce	Cu	Ci	Ca	Cie	C	Co	Cua				
37	Pe	Pu	Pi	Pa	Pie	P	Po	Pua				
38	xe	xu	xi	xa	xie	x	xo	xua				
39	Xe	Xu	Xi	Xa	Xie	X	Xo					
40	fe	fu	fi	fa	fie	f	fo	fua				
41	pe	pu	pi	pa	pie	p	po	pua				

Source: <http://www.geezsoft.com/gg2man/Table.htm>

Available Fonts

One of the advantages of having standards for the encoding of Ge'ez letters is that fonts from a diversity of unrelated font developers can all be used with the standard encoding systems. Since the Unicode standards for Ge'ez are still fairly new there are only a few fonts available, but more are in development. Currently available Unicode compliant fonts include:

1. GF Zemen Unicode, URL: <ftp://ftp.ethiopic.org/pub/fonts/TrueType/gfzemenu.ttf>
2. Ethiopia Jiret, URL: <http://www.senamirmir.com/projects/ethiopic/ethiopic.html>
3. Code2000, URL: <http://home.att.net/~jameskass/>

For Tfanus/Yada the font that is used is GeezTimesNew, while the main font for Phonetic Systems/GeezGate is GeezTypeNet (at URL: <http://www.geezsoft.com/geeztntnet.ttf>)

Software Design

Background: Non-language Specific

Windows 2000 is designed from the ground up to internally handle Unicode strings. All of the core functions for creating windows, displaying text, manipulating strings, and so forth are implemented using Unicode. This is not the case in Windows 95 and 98, where almost every function uses ANSI strings. There are some functions such as TextOutW and MessageBoxW that are capable of handling Unicode in Windows 9x. However, the few functions that do handle Unicode are not reliable. Some of them do not work with certain fonts, some of them corrupt the heap, and some of them crash printer drivers (Richter 1999). It is possible to design one's own Unicode text editor using the TextOutW function, but because of the limited number of operable Unicode functions for Win9x, overall functionality would probably be severely limited. A good solution to the problem of getting Unicode input into Windows 95 and 98 would be to have a program intercept keyboard input, transform that input to Unicode characters, and then pass those characters on to the destination application such as Microsoft Word. That brings us to two important issues: How do we intercept keystrokes and how do we change them to Unicode?

Before answering these questions, we need a little background in the Windows Operating System architecture. In Windows programming, practically everything that we deal with is an object. Programs are comprised of windows, windows within windows, buttons, scroll bars, controls, and menus to name a few. All of these objects need to communicate back in forth with each other, with the user, and with the operating system. The communication system of Windows programming is orchestrated through the sending and receiving of messages asynchronously. Every Windows program contains a main message loop that runs continuously through the execution of the code to receive, translate, and dispatch messages. These messages could be coming from other processes, from the operating system, or from the process itself. The messages that are routed from window to window are nothing more than integers that have been defined to have special meaning in the Windows OS. When a window is created in a program, that window is assigned a unique identifier, called a handle. In order to send a message to a specific window, one specifies that window's handle. For example, when a user clicks on the close button of a window to end that application, a WM_CLOSE message is sent to that window's message loop. The application can choose to process that message and call a function to end the program or it can ignore the message. Any messages that are not explicitly handled by a process are passed on to its default message loop for final processing. Continuing with the example, if a process did not process the WM_CLOSE message, then the application would still end, because the default message procedure receives the message and calls a function to end the process itself.

This brings us to an interesting question: Can we modify the way in which the default message loop processes certain messages, and the answer is yes! Normally, if we wanted to create a text editor, then we would create a window that is an edit control, and the edit control window would handle the displaying of text without any work on our part. Every time a user presses a key while running the editor, a WM_CHAR message would be processed by the default message procedure causing that character to appear in the text editor's window. However, by using a technique known as subclassing, we can intercept and process messages sent to a particular window before the window has a chance to process them. Subclassing basically tells the main message loop to pass certain messages to a programmer-defined message loop for processing, which in turn passes them on to the default message procedure. If we were to subclass the main message loop of the editor, then when a user pressed a key, we would get a WM_CHAR message in our own message loop. We could, at this point, change the character code to 88 (ASCII 'X') from whatever it was before, and then pass that on to the default message procedure, so that every time the user pressed any key, an ASCII 'X' would be outputted. Here is where the distinction between Windows 2000 and 9x comes to bare. In Windows 2000, instead of changing any character code to 'X', it could be changed to any 16-bit Unicode value, since Windows 2000 is designed to internally handle Unicode. However, in Windows 9x this is not the case. Changing the 'X' to Unicode in 9x does not result in the output of Unicode, but in two separate bytes if multi-byte characters are defined or a single byte if they are not defined. This identifies the problem of getting Unicode characters into Windows 95 and 98, but how do we resolve it?

Fortunately, as early as Windows 95, Unicode support was built into the clipboard—that mechanism whereby we can copy, cut, and paste data from one application to another. If one was editing in Microsoft Word and copied a paragraph of text, that text goes to the clipboard in some sixteen different formats, one of which is the CF_UNICODETEXT format. The reason for the numerous clipboard formats is to allow the widest possible selection of formats for other applications to choose from when we attempt to paste that text somewhere else. Each application determines what clipboard formats it will read, and each determines what formats they will use for placement of their data on the clipboard. One can copy text from Word and paste that text into Notepad because Word formatted that text with the CF_TEXT format and Notepad is capable of reading that same format. If Word had only formatted that text with the HTML format, then it would not have pasted into Notepad, since Notepad cannot read the HTML format. We have not yet discovered how we intercept keystrokes, but we have answered the question of how we are able to achieve Unicode input into Windows 95 and 98—through the clipboard. This is a good place to note that many applications in Windows 95 and some in 98 do not read the CF_UNICODETEXT format, and many of these same applications do not read Unicode text at all if formatted in either the HTML or RTF formats. It is up to the individual application to support Unicode in order for Unicode text to be pasted into that application via the clipboard.

This brings us to the following question: If a user is typing in Microsoft Word, how do we intercept their keystrokes and paste in Unicode text from the clipboard instead? The answer is a hook. In the Windows OS, each process gets its own private address space so that the failure of one process does not cause other processes to crash. But situations do arise where it is necessary to break through the process boundary walls to access the address space of some other process (Richter 1999), and intercepting keystrokes bound for another process is the perfect example. A hook is a Win32 API function defined by Windows that allows one to intercept messages bound for the default message procedure in the address space of a process other than your own. It is like the

subclassing example mentioned afore, only it takes place not in your own address space, but in another's. In order for a hook to work, it must reside in a DLL (Dynamic-Link Library). A DLL usually consists of a set of autonomous functions that any application can use. Before an application can call functions in a DLL, the DLL's file image must be mapped into the calling process's address space. This is known as DLL injection. To the process in which the DLL's code and data resides, it merely looks like additional information that happens to be in the process's address space(Richter 1999).

Language-Specific Implementation

The UniGeez user interface window provides a means to map the UniGeezTransliterator DLL's file image into the address space of all processes currently running in the system. By pressing the 'ON' button, the user interface references functions contained in the DLL, which causes the loader to implicitly load (and link) the DLL into the address space of all current processes. Likewise, by pressing the 'OFF' button, the DLL is unloaded from all the same address spaces. Once the DLL has been injected into the address space of all processes, then no matter which application a user may be using, the hook code contained within the DLL, and thus within the application, is filtering the messages that will be processed by that process's message loop. By using the WH_GETMESSAGE hook, not only can we look at the messages that produce characters, we can also modify them. Again, it is like the subclassing example, where we have our own message loop caught between the main message loop and the default message loop. At this point, we can prevent messages from being processed at all by setting them equal to zero. Quit a powerful little tool! Let it be noted, however, that system hooks are draining on system resources, and they can be potentially harmful if not properly utilized. Furthermore, hooks are not capable of intercepting all messages. Different types of hooks can intercept different messages, and some can modify messages while others can only look at them. The WH_GETMESSAGE hook can intercept and modifying character messages such as WM_KEYDOWN, WM_KEYUP, and WM_CHAR.

Before jumping into a full discussion on Unicode input in Windows 9x, we need to talk about the raw input thread (RIT). When the Windows OS initializes, it creates a special thread known as the raw input thread. It also creates a queue called the system hardware input queue (SHIQ). The RIT and SHIQ together form the heart of the hardware input model. When a user presses and releases a key, a mouse button, or moves the mouse, the device driver for the hardware device adds the event to the SHIQ. The SHIQ in turn, wakes up the RIT, and the RIT extracts the event from the SHIQ and translates the event into the appropriate message. The translated message is then appended to the appropriate thread's virtualized input queue (VIQ), and the RIT goes back to wait for more messages to appear in the SHIQ. For mouse events, the RIT appends messages to the VIQ of the window under the mouse cursor. Keyboard messages are appended to the VIQ of the foreground thread. The foreground thread belongs to the window that the user is interacting with, and this window is in the foreground with respect to the other windows that may be opened(Richter 1999).

All this is mentioned for two reasons. One, even though we are running a system wide hook, since we are only seeking out character messages, we are basically only dealing with the foreground window at any one time. However, the system hook is necessary, so that when the foreground window changes, we can immediately start intercepting the messages of the new foreground window as well. Two, in order to achieve Unicode output in Windows 9x, it is necessary to synthesis keystrokes in the keyboard driver to affect Unicode output. Understanding how keystrokes are routed as messages to the foreground window is a good basis for the notion of synthesizing keystrokes.

With the ground work laid, let us proceed with an example of how Ge'ez Unicode output is achieved in existing applications in the Windows environment. A user starts by running the UniGeez executable and pressing the 'ON' button. Then Word is opened (or could have already been open), and they type the 'h' character. The DLL's code is now inside of Word's address space, and since the RIT is connected to the window with cursor focus, the 'h' message is routed to Word's main message loop. At this point, the DLL's code (the hook) intercepts the character message and processes it. The DLL utilizes a finite state machine that begins in a nothing state, and then changes to a state that says it just received an 'h'. It determines that 'h' has a Unicode Ge'ez equivalent (0x1205), it formats that Unicode character using RTF, it places that format on the clipboard, and it synthesizes CTRL-V in the keyboard driver. The paste sequence is synthesized by saying the CTRL-key went down, the V-key went down, the V-key went up, and the CTRL-key went up. This past message is routed to Word because it is the foreground window, and Word carries out the paste message as if the user had pressed CTRL-V. Word looks to the clipboard, reads the one Unicode character formatted in RTF, and pastes that Ge'ez Unicode character into its window next to the cursor. Word and WordPad use the RTF clipboard format; Excel, PowerPoint, and Access use the Unicode clipboard format; and FrontPage uses the HTML clipboard format. Basically, all three formats, RTF, HTML, and Unicode are capable of formatting Unicode characters for the clipboard, but different applications work best with specific clipboard formats.

So, now we have the Ge'ez Unicode character, 0x1205, in Word's window, and the finite state machine is in the 'h' state. Next the user types 'a'. The state machine enters the 'ha' state (the last character was a 'h' and the current character is 'a'). By virtue of the fact that we are in the 'ha' state as opposed to the 'nothing' state, we know that the last glyph printed to screen was 0x1205. Since 'ha' is a different glyph from 'h', a flag is set to force a backspace. This means that we emulate a keystroke for the backspace event in the keyboard driver so that the previous 'h' (0x1205) is erased. Then, 'ha' (0x1203) is formatted for the clipboard, placed on the clipboard, and a paste sequence is issued to output the Unicode glyph 0x1203. The finite state machine recognizes that 'ha' is the end of a state and its state becomes 'nothing', waiting for more input. Recapping what has taken place, the user typed 'h' and 0x1205 was pasted in; the user typed 'a' and a backspace erased 0x1205, and 0x1203 was pasted in from the clipboard in its place. The same process continues for all the other Ge'ez glyphs. Therefore, by DLL injection/API hooking and pasting Unicode text from the clipboard, Unicode output is achievable in Windows 95 and 98, provided that the destination application supports Unicode. The same method also works in Windows NT and 2000, though is not altogether necessary because of their already existing support of Unicode.

Converting Non-Standard Files to Unicode

Before the advent of Unicode, languages that had in excess of the standard 256 ASCII characters had to rely on other methods to get all of their glyphs to display. Keyboard layouts were developed for certain languages, language specific versions of the Windows OS were marketed, multi-byte code pages were implemented in the Windows architecture, and now even TrueType fonts are used as a means to map out the keyboard beyond the standard Latin code page. Unfortunately, though, for the Ge'ez language, there are no keyboard layouts or multi-byte code pages defined that would provide a consistent way to encode their glyphs. Various companies and individuals set out with the best of intentions to design software that would be capable of producing all of the Ge'ez glyphs for various platforms. However, no uniform standard was applied, and so today, there exists some seventy different encoding standards for the Ge'ez alphabet. Naturally, it follows then, that since we have the Unicode standard and since we have a method for inputting its glyphs in Windows, that we would want to convert files from non-standard encoding to Unicode.

Two of the more popular and widely used encodings, Tfanus and Phonetic, were targeted for conversion to Unicode. The first hurdle that had to be crossed was coming up with conversion tables for converting Tfanus and Phonetic to Unicode. The Libeth project had initially provided many of the conversion

sequences, but some were simply out-dated or incorrect. The second obstacle was coming up with a way to parse the various file formats used by Microsoft (doc, xls, ppt, mdb, rtf, htm, and txt). It would be simple enough to parse rtf, htm, and txt files, but how about the others, and in particular, Word's doc files and Excel's xls files. Microsoft's file formats are proprietary, and though many have attempted to reverse engineer them, the attempts were either largely unsuccessful, or became proprietary themselves. Once again, we looked to the clipboard for a solution.

Even though Microsoft may annoy us from time to time, there is a lot of robustness and functionality built into their design of the OS (some of it probably unintentional), so that if you work hard enough and think about it long enough, you will find a solution. The clipboard is one of those seemingly little features that we often take for granted, but it is an extremely powerful tool. As mentioned earlier, whenever you copied data from an application, it goes to the clipboard in numerous different formats, such as HTML, RTF, Unicode, plain text, bitmaps, metafiles, and many more. In all of Microsoft's Office products (except FrontPage), when you copy text from one of their applications, it is exported to the clipboard as RTF along with other formats. So, even though one might be word processing in Word and that file will probably be save as a doc file, if one copies that text to the clipboard, it could be viewed as RTF. The same holds true for spreadsheets in Excel, tables in Access, and slides in PowerPoint.

To this end, a conversion utility was developed that would run in conjunction with the UniGeez Transliterator to convert Tfanus and Phonetic files to Unicode. It also supports converting back from Unicode to either Tfanus or Phonetic. Even though Unicode may be the best solution to language modernization in the realm of binary data transmission, people are still reluctant to switch from the old to the new, so backwards compatibility was considered important to support as many users as possible. The best way to describe how the conversion utility works is probably to walk through an example.

The user starts by running the UniGeez Transliterator and pressing the 'ON' button. Then they can go to Word to convert doc, rtf, html, and txt files. After opening a Tfanus or Phonetic file (or combination of both), they simply press F6. Since the hook is running inside of Word, we can synthesis a CTRL-A (select-all) and CTRL-C (copy) message in the keyboard driver, so that the entire file is selected and copied to the clipboard. Another option is for the user to manually highlight portions of text and press SHIFT + F6, which simply issues a CTRL-C sequence to copy just that selected portion to the clipboard. All the user ever sees is their file being highlighted, and then a blink-of-the-eye later, the appropriate portions of the file are in Unicode!

Behind the scenes, after the user has pressed F6, the file is retrieved from the clipboard in the RTF format. If the user is word-processing in either Word or WordPad, then the converted file will be pasted back using the RTF format, otherwise the Unicode clipboard format is used. But for this example, let's consider that the user is in Word. By parsing the RTF format, we can isolate actual text from formatting information. The fonttbl keyword in RTF signifies the font definitions group, and inside that group, 'f' plus some number followed by a font name creates a mapping between the font and numeric value. For example, {fonttbl{\f12 Times New Roman}}, establishes a relationship between f12 and Times New Roman. When the f12 keyword is encountered outside of the 'fonttbl' group, it means that all the text within its group, or until another 'f' keyword is encountered, should be displayed using the Times New Roman font. In terms of converting Phonetic and Tfanus to Unicode, the RTF parser seeks out GeezTypeNet and GeezTimesNew in the font definition group and changes them to GF Zemen Unicode. Of course, this is not enough to convert to Unicode. The parser also seeks out those groups that correspond to those two fonts and changes the actual encoding from one to three byte sequences to 16-bit Unicode values.

The conversion tables for converting Tfanus and Phonetic to Unicode are stored as Excel files for easy manipulation, reading, and presentation. The Excel files can then be saved as tab-delimited text, where each column is separated by a tab, and then that text file is feed to a program for parsing. The program navigates the rows by the tabs that separate each column and produces a file of tokens for the lex scanner.

Example from the Phonetic conversion table in Excel format:

```
hu 0x1201 40 e9
```

Result after running through program for parsing into a lex token:

```
<PHONETIC>[\x40][\xe9] { return hu; }
```

With 'hu' enumerated to equal 0x1201, the lex statement reads: for Phonetic, 0x40 followed by 0xE9 converts to 0x1201. So, as you have already guessed, the conversion process is handled by the lex scanner, based on the conversion tables. The whole process is automated to promote efficiency. To make a change in how the utility converts text, one simply updates the Excel file, saves it as tab-delimited text, runs it through the program, runs the product of the program through the lex executable, takes the lex.yy.c file and places it with the conversion utility, and recompiles. In less than a minute, the conversion utility can work off of any number of new definitions in either the Phonetic or Tfanus conversion tables.

Therefore, after changing the font definition mapping and after running the corresponding text through lex, we have a new RTF format to place on the clipboard. Finally, a CTRL-V event is created in the keyboard driver, and the converted data is pasted back into the application from which it was copied. Because the user is working in Word (or WordPad), the converted file can be pasted back in using the RTF format and all text formatting is preserved. When you have text that is in a different font size or when you have text that is bold, italic, underlined, etc., after that text is converted to Unicode, it retains those formatting properties. Back converting from Unicode to either Phonetic or Tfanus, happens in much the same fashion. The definitions still come from the same conversion tables as before, and, instead of using lex, direct access arrays are used to map Unicode to multiple byte sequences.

There is a little "twist" to the conversion process. Whereas Word can import and export text using the RTF format, Excel, PowerPoint, and Access can only export text as RTF; they cannot import text as RTF. In order to get Unicode characters into these applications, text must be pasted using the CF_UNICODETEXT format. The conversion process start off by detecting which application the user is currently in. If the user is in Word or WordPad, then the afore mentioned process is followed ending in the RTF paste. Otherwise, a slightly different approach is taken. The basic premise behind the alternative approach is this: regardless of the file format, if just the text is extracted from two different formats, then there should be a one-to-one correspondence between the bytes from each of the two files (same data, different formats). Let's apply that premise to converting a Excel file.

The user has UniGeez 'ON', opens a Tfanus file in Excel, presses F6, and the entire file is copied to the clipboard. We retrieve the file with the RTF format and start parsing. However, since we are dealing with a Excel file that requires a Unicode paste, we only parse the RTF to map out what bytes needs converting. If you look at the section were the actual data resides in RTF, that section can be divided into groups based on the font that the group is using. When the RTF is parsed for creating a Unicode map, small lists of information are stored in a data structure. Each list tells how many bytes are in the group, whether the group requires conversion, and what font is associated with the group. We said previously that we copied the RTF format from the clipboard, but actually, at the same time we also retrieved the Unicode format from the clipboard. Keep in mind that Unicode for the ASCII characters uses the same ASCII codes, so that if we look at the Unicode version of the Phonetic encoding it is exactly the same as if we had looked at its text version, since the high-byte is set to zero. It is important to note that the Unicode clipboard format contains no formatting information. That is, if a cell contains characters that are in boldface, copying the contents of that cell to the clipboard and viewing it as Unicode only reveals the character codes—the Unicode has no idea that it is boldface.

The next step in converting the Excel file is to apply the RTF map to the Unicode format. It will probably be best to work from an example. Let us suppose that row 1 has 25 bytes in the Arial font, row 2 has 21 bytes in the GeezTimesNew font, and row 3 has 7 bytes in Times New Roman. The RTF map for this would resemble the following:

```
map[0]: 25 bytes, do not convert, Arial
map[1]: 21 bytes, convert, GeezTimesNew
map[2]: 7 bytes, do not convert, Times New Roman
```

The RTF map says that we have a total of 53 bytes, so the Unicode format should also have 53 pairs (remember Unicode is always 2 bytes, and by pairs I mean one low byte and one high byte). The first 25 pairs from the Unicode format are skipped over. Starting with pair 26, we recognize that this pair along with the next 20 pairs need converting. So, all 21 pairs are passed through the lex scanner, and a lesser number of Unicode values is the result. The Unicode output from lex is then used to replace the entire second group. The last 7 bytes do not require conversion, so they are left alone. Now we have a new Unicode format with the first and last group left alone, but with the middle group converted from Tfanus to Unicode. Back converting is even easier; no RTF map is even needed! We simply seek out those Unicode values that are within the Ge'ez range (0x1200 – 0x137C), and convert them to either Tfanus or Phonetic through a direct access array. Although with a Unicode paste we cannot do anything about the formatting, we can pick out the right characters to convert.

FrontPage does not use the RTF format; it uses HTML. Since HTML files can be opened in Word, that is probably the best place to convert them. The conversion utility can convert files without the RTF map, and it does so in FrontPage and NotePad based on user settings. See the help files for the conversion utility for more detailed information about converting specific file types or about converting files in specific applications.

Software Licensing

We felt it was important to make the UniGeez software and fonts available as widely as possible, while insuring that incompatible proprietary variants are not created. Therefore we decided release the code as Open Source under the GNU General Public License, the Open Source License used by Linux (see <http://www.gnu.org/copyleft/gpl.html>).

The license grants a lot of rights to the possessor of the software, but also imposes certain restrictions. Anyone in possession of the software can use it, copy it, study it, modify it, sell it, or give it away for free. (In full compliance with this license there is at least one computer company in Eritrea copying and selling the software, as well as charging for installations and support.)

However, the license does require that the complete source code for the software (and any modifications to it) be supplied with each copy *and* that an unchanged copy of the license agreement accompany the software.

This means that a business or individual cannot appropriate our work by incorporating the software into a program and then adding proprietary extensions that are incompatible with other versions of the code (the infamous Microsoft strategy of "embrace and extend", used to create proprietary versions of public domain software and protocols). While anyone is free to enhance or extend the software (even in a fashion incompatible with our release of the software), they cannot prevent others from incorporating those extensions. Thus, if the extensions are useful and valuable they can be incorporated into our, or others, release of the software.

There is a certain tension between our goals. If the software were placed in the public domain, such that anyone was free to do what they wanted with it including incorporating it into proprietary programs (such as a popular text editor) then the software would likely be more widely distributed. But there would be a real risk of the proprietary software making non-standard and proprietary modifications/extensions to our software or character encodings that would limit the ability of user to freely exchange documents that use the encodings. So we chose to adopt an approach that we hope will maximize the distribution and usefulness of the software.

While it's unlikely that a license agreement based on U.S. Copyright Law would be enforced by a court in Eritrea, the intent of the license is plain. And if any infringing software or use of the fonts was brought to the U.S., than U.S. courts could be involved.

For a more general discussion of software licensing and Free Software can be found at the GNU website: <http://www.gnu.org/philosophy/free-sw.html>.

Conclusions

As Eritrea's computer information and communications infrastructure develops, it is necessary to strive for continuing improvement in efficiency, productivity, and accessibility. Initially multi-lingual computer applications in Eritrea provided Tigrigna language and Ge'ez capabilities by replacing Western glyphs with Ge'ez glyphs using non-standard encodings designed by individual software developers. But since October 1998, an international standard for Ge'ez characters has been in effect, while the software and facilities for standard-compliant Ge'ez has lagged. The Eritrean Technical Exchange Project in collaboration with the Computer Science Department of Lynchburge College has filled the need for standard-compliant Ge'ez software by developing UniGeez, keyboard input software that runs under Windows. The software runs with virtually all applications that support the Unicode standard and the UTF-8 character encoding scheme. The software has been developed under an open-source license in order to make it transparent and so that others may make improved variants. We believe standard compliant Ge'ez will be dominant in Eritrea and Ethiopia over the long term, and hope that UniGeez will make the transition to standard compliant easier and more rapid. Future plans for UniGeez development include several elements:

- Inclusion and testing of conversion utilities that can convert documents between Unicode, Phonetic Systems, and Tfanus encodings.
- Inclusion of standard compliant Arabic capabilities.
- Allowance for a choice of keyboard mappings or input schemes.
- Development of a greater diversity and variety of fonts.
- Enhanced training in multi-lingual computer use and improved distribution of UniGeez or improved variants.
- Expansion of multi-lingual capabilities to a diversity of operating systems and Internet applications (e.g. database applications).

We hope that eventually the continued development and improvement of standard compliant multi-lingual computer capabilities will be taken over more completely by the private sector and the growing community of Eritrean computer professionals. We also seek private/non-profit partnerships to further develop multi-lingual computer infrastructure in the optimum Eritrean public interest. We believe that with the enhanced, productivity, accessibility, and

efficiency that comes with high quality standard compliant multi-lingual infrastructure, Eritrea's economic development will be greatly enhanced.

REFERENCES

Czyborra, Roman, *The Global Character Set Unicode in the Unix Operating System*, Draft Master's Thesis, November 30, 1998, URL: <http://czyborra.com/diplom/abgabe.html>

Richter, Jeffrey, *Programming Applications for Microsoft Windows*, 4th edition, Microsoft Press, 1999.

Unicode Inc., *Unicode Standard 3.0*, Ethiopic, Range 1200 - 137F, URL: <http://www.unicode.org/charts/PDF/U1200.pdf>

Yacob, Daniel, *LibETH Project Homepage*, January 31, 2001, URL: <http://libeth.sourceforge.net/>

Yacob, Daniel, *Frequently asked questions about SERA*, Abyssinia Cybergateway, December 24, 1996, URL: <http://www.abysiniacybergateway.net/fidel/sera-faq.html>

APPENDIX: Encoding and Font Tables

Table of SERA Transliterations

	1	2	3	4	5	6	7	8	9	10	11	12
1	he	hu	hi	ha	hE	h	ho					
2	le	lu	li	la	lE	l	lo	lWa				
3	He	Hu	Hi	Ha	HE	H	Ho	HWa				
4	me	mu	mi	ma	mE	m	mo	mWa				
5	`se	`su	`si	`sa	`sE	`s	`so	`sWa				
6	re	ru	ri	ra	rE	r	ro	rWa				
7	se	su	si	sa	sE	s	so	sWa				
8	xe	xu	xi	xa	xE	x	xo	xWa				
9	qe	qu	qi	qa	qE	q	qo	qWe	qW	qWi	qWa	qWE
10												
11	Qe	Qu	Qi	Qa	QE	Q	Qo	QWe	QW	QWi	QWa	QWE
12	be	bu	bi	ba	bE	b	bo	bWa				
13	ve	vu	vi	va	vE	v	vo	vWa				
14	te	tu	ti	ta	tE	t	to	tWa				
15	ce	cu	ci	ca	cE	c	co	cWa				
16	`he	`hu	`hi	`ha	`hE	`h	`ho	hWe	hW	hWi	hWa	hWE
17	ne	nu	ni	na	nE	n	no	nWa				
18	Ne	Nu	Ni	Na	NE	N	No	NWa				
19	e	u	i	a	E	I	o	ea				
20	ke	ku	ki	ka	kE	k	ko	kWe	kW	kWi	kWa	kWE
21	Ke	Ku	Ki	Ka	KE	K	Ko	KWe	KW	KWi	KWa	KWE
22												
23												
24	we	wu	wi	wa	wE	w	wo					
25	`e	`u	`i	`a	`E	`I	`o					
26	ze	zu	zi	za	zE	z	zo	zWa				
27	Ze	Zu	Zi	Za	ZE	Z	Zo	ZWa				
28	ye	yu	yi	ya	yE	y	yo	yWa				
29	de	du	di	da	dE	d	do	dWa				
30	De	Du	Di	Da	DE	D	Do	DWa				
31	je	ju	ji	ja	jE	j	jo	jWa				
32	ge	gu	gi	ga	gE	g	go	gWe	gW	gWi	gWa	gWE
33												
34	Ge	Gu	Gi	Ga	GE	G	Go					
35	Te	Tu	Ti	Ta	TE	T	To	TWa				
36	Ce	Cu	Ci	Ca	CE	C	Co	CWa				
37	Pe	Pu	Pi	Pa	PE	P	Po	PWa				
38	Se	Su	Si	Sa	SE	S	So	SWa				
39	`Se	`Su	`Si	`Sa	`SE	`S	`So					

40	fe	fu	fi	fa	fE	f	fo	fWa
41	pe	pu	pi	pa	pE	p	po	pWa

Table of Phonetic Systems (GeezGate) Character Encodings (in Hexidecimal)

	1	2	3	4	5	6	7	8	9	10	11	12
1	\x40	\x40\xe9	\x41\xf0	\x41	\x41\xf4	\x42	\x43					
2	\x44	\x44\xe9	\x44\xef	\x45	\x44\xed	\x46	\x44\xf7	\x45\xfb				
3	\x47	\x47\xe9	\x47\xef	\x48	\x47\xed	\x49	\x4a	\x48\xfa				
4	\x4b	\x4b\xe9	\x4c\xf0	\x4c	\x4c\xf4	\x4d	\x4e	\x4c\xfc				
5	\x4f	\x4f\xe9	\x50\xf0	\x50	\x50\xf4	\x51	\x52	\x50\xfb				
6	\x53	\x54	\x55	\x56	\x57	\x58	\x59	\x5a				
7	\x5b	\x5b\xe9	\x5b\xef	\x5c	\x5b\xf2	\x5d	\x5e	\x5c\xfa				
8	\x5f	\x5f\xec	\x5f\xf0	\x60	\x5f\xf4	\x61	\x62	\x60\xfc				
9	\x63	\x63\xeb	\x63\xf1	\x64	\x63\xf5	\x65	\x66	\x63\xf8	\x63\xff	\x63\xf9	\x63\xfc	\x63\xfe
10												
11	\x63\x68	\x63\x68\xeb	\x63\x68\xf1	\x64\x68	\x63\x68\xf5	\x69	\x6a	\x63\x68\xf8	\x63\x68\xe8	\x63\x68\xf9	\x63\x68\xfc	\x63\x68\xfe
12	\x6b	\x6b\xe9	\x6b\xef	\x6c	\x6b\xf2	\x6d\x6b	\x6e	\x6c\xfa				
13	\x6b\x7a	\x6b\x7a\xe9	\x6b\x7a\xef	\x6c\x7a	\x6b\x7a\xf2	\x6d\x6b\x7a	\x6e\x7a	\x6c\x7a\xfa				
14	\x6f	\x6f\xeb	\x6f\xf1	\x70	\x6f\xf5	\x71	\x72	\x6f\xfc				
15	\x6f\x68	\x6f\x68\xeb	\x6f\x68\xf1	\x70\x68	\x6f\x68\xf5	\x75	\x76	\x6f\x68\xfc				
16	\x77	\x77\xe9	\x77\xf0	\x78	\x77\xf3	\x79	\x6d\x7e	\x77\xf7	\x77\xff	\x77\xf9	\x77\xfb	\x77\xfd
17	\x7b	\x7b\xe9	\x7b\xf0	\x7c	\x7b\xf3	\x7d	\x7e	\x7c\xfc				
18	\x8f	\x8f\xec	\x8f\xf1	\x81	\x8f\xf5	\x82	\x83	\x81\xfc				
19	\x84	\x84\xe9	\x84\xef	\x85	\x84\xed	\x86	\x87	\x88				
20	\x89	\x89\xe9	\x89\xef	\x8a	\x89\xf2	\x8b	\x8c	\x89\xf7	\x89\xe8	\x89\xf9	\x8a\xfa	\x89\xfd
21	\x89\x8d	\x89\x8d\xe9	\x89\x8d\xef	\x8a\x8d	\x89\x8d\xf2	\x8b\x8d	\x8c\x8d	\x89\x8d\xf7	\x89\x8d\xe8	\x89\x8d\xf9	\x8a\x8d\xfa	\x89\x8d\xfd
22												
23												
24	\x91	\x92	\x67\xf1	\x67	\x67\xf5	\x91\xe9	\xc8					
25	\x95	\x95\xe9	\x96\xf0	\x96	\x96\xf3	\x97	\x98					
26	\x73	\x73\xe9	\x73\xef	\x9a	\x73\xf2	\x9b	\x9c	\x9a\xfa				
27	\x9d	\x9d\xee	\x9d\xf1	\x90	\x9d\xf5	\x9f	\xa1	\x90\xfc				
28	\xa2	\xa3	\xa4	\xa5	\xa6	\xa7	\xa8					
29	\xa9	\xaa\xe9	\xaa\xef	\xaa	\xab	\xa9\xf6	\xac	\xa9\xfa				
30	\xa9\xc2	\xaa\xc2\xe9	\xaa\xc2\xf0	\xaa\xc2	\xab\xc2	\xa9\xc2\xf6	\xac\xc2	\xa9\xc2\xfa				
31	\xad	\xae\xec	\xae\xf0	\xae	\xaf	\xb0	\xb1	\xad\xfa				
32	\xb2	\xb2\xe9	\xb2\xef	\xb3	\xb2\xf3	\xb4	\xb5	\xb2\xf7	\xb2\xe8	\xb2\xf9	\xb6	\xb2\xfd
33												
34	\xb2\x7a	\xb2\x7a\xe9	\xb2\x7a\xef	\xb3\x7a	\xb2\x7a\xf3	\xb4\x7a	\xb5\x7a					
35	\xb7	\xb7\xe9	\xb7\xef	\xb8	\xb7\xf2	\xb9	\xba	\xb7\xfa				
36	\xbb	\xbb\xec	\xbc	\xbd	\xbe	\xbf	\xc0	\xc1				
37	\xc3\xc2	\xc3\xc2\xe9	\xc3\xc2\xef	\xc4\xc2	\xc3\xc2\xed	\xc3\xc2\xf6	\xc5\xc2	\xc4\xc2\xfa				
38	\xc3	\xc3\xe9	\xc3\xef	\xc4	\xc3\xed	\xc3\xf6	\xc5	\xc4\xfa				
39	\x95\xc6	\x95\xc6\xe9	\xc7\xef	\xc7	\xc7\xf3	\x97\xc6	\xc9					
40	\xca	\xcb	\xcc	\xcd	\xce	\xcf	\xd0	\xcd\xfc				
41	\xd1	\xd1\xee	\xd1\xf1	\xd2	\xd1\xf5	\xd3	\xd4	\xd1\xfc				

Table of Tfanus (Yada) Character Encodings (in Hexidecimal)

	1	2	3	4	5	6	7	8	9	10	11	12
1	\xc0	\xc0\xf1	\x40\xd4	\x40	\x40\xd4	\x41	\x42					

