Multi-Language Character Sets

Bob Balaban, President
Looseleaf Software, Inc.
http://www.looseleaf.net

What They Are, How to Use Them

Click here to add clip art

Agenda

- Speaker introduction
- Background: terminology
- History of electronic character sets
- The need for multi-language character sets
- LMBCS - Lotus MultiByte Character Set
  - How it works
  - Why no one else uses it

Agenda - 2

- Unicode
  - How it works
  - Why everyone uses it
- Software for multi-language character sets
  - LMBCS
  - Unicode
  - Java, C, LotusScript
- Web applications and MLCS

Speaker Introduction

- Master's degree in Chinese History/ Anthropology
- Professional software developer since 1978
- Engineer at Lotus Development Corp. 1987-97
- Reviewer of original LMBCS specification (1988)

Speaker - 2

- Team leader for adding LMBCS/Kanji support to 123/G (os/2)
- Developer on Notes/Domino 1993-97
  - Author of LotusScript "back-end classes"
  - Author of Java APIs for back-end classes
- Founded Looseleaf Software, Inc., 1997
  - Custom development, training, architecture/design, consulting
  - Notes/Domino, J2EE, Groove

Terminology

- Textual characters on a computer are really numbers
  - Like everything else
- The numbers, when representing text, are mapped to symbols on a screen
  - or on paper
- The number of symbols you can display depends on the number of bits assigned to the "character set", or "code space"

Terminology - 2

- One "character" is a "code point"
  - Just a number
- The screen symbol is the "glyph"
- The "font" is the style in which the glyphs are displayed
  - serif, sans serif, etc.
- The mapping of numbers to code points defines a "character set", or "code page"

Terminology - 3

- What is a "native character set"?
- The default character set used by the operating system
- Many OS's can handle multiple character sets (Ascii, CP850, etc.)
- Many OS versions depend on a "locale" specification
  - E.g., CP932 in Japan, CP850 in NAmerica

Terminology - 4

- All mappings are essentially arbitrary
- But some have been agreed upon as standards

A = 1
B = 2
...
Z = 26

A possible mapping

A = 17
B = 18
....
K = 99
....
Z = 102

Another possible mapping

Terminology - 5

- What are "control codes"?
- Generally, non-printing
  - No visible representation, no glyph
- Invented to manage early printer/ terminal devices, and communication protocols
  - Ack/Nak
  - Carriage return, newline, tab, formfeed
  - Bell

Background: History of electronic character sets

- Original character set was Binary Coded Decimal (BCD)
  - IBM
  - 64 characters, 6 bits
  - Known later as "SIXBIT ASCII"
- Developed for IBM punch cards
- Upper case letters, digits, punctuation

Background - 2

- When IBM created the System 360, they extended BCD
  - 1965
  - EBCDIC
  - Extended Binary Coded Decimal Interchange Code
  - 256 code points
  - 64 control codes
  - Upper and lower case!
  - Some slots were empty, customers complained about wasted memory!

IBM EBCDIC, ca. 1965

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	A.	B.	C.	D.	E.	F.
.0	NUL	DLE			SP	&	-						{	}	\	0
.1	SOH	DC1					/		a	j	~		A	J		1
.2	STX	DC2		SYN					b	k	s		B	K	S	2
.3	ETX	DC3							c	l	t		C	L	T	3
.4									d	m	u		D	M	U	4
.5	HT		LF						e	n	v		E	N	V	5
.6		BS	ETB						f	o	w		F	O	W	6
.7	DEL		ESC	EOT					g	p	x		G	P	X	7
.8		CAN							h	q	y		H	Q	Y	8
.9		EM						`	i	r	z		I	R	Z	9
.A					¢	!	¦	:								
.B	VT				.	$	,	#								
.C	FF	FS		DC4	<	*	%	@								
.D	CR	GS	ENQ	NAK	(	)	_	'								
.E	SO	RS	ACK		+	;	>	=								
.F	SI	US	BEL	SUB			^	?	"							

EBCDIC and Beyond

- The most popular character set through the 1970s
  - Until the PC was born
- IBM used 8 bits because the word size on S360 was 32 bits
- IBM machines (other than PCs) still use EBCDIC
  - There are a few variants, some code points may be different

Beyond - 2

- Note the strange ordering, where the alphabetic sequences are interrupted
- Makes for lots of fun when writing string comaprison/sorting code!
  - Or converting between upper and lower case
- Note, no accented characters
  - Or non-Latin glyphs

The Rise of ASCII

- American Standard Code for Information Interchange
  - Or something like that
- Network bandwidth was very expensive, people wanted to save money and utilize 7-bit channels
- And wanted to conserve on memory, eliminate all that "wasted" space in EBCDIC

ASCII - 2

- Adopted by PC OSs (DOS), and by Unix systems
- 32 control codes, punctuation, digits, upper and lower alphabetics (0 is special)
  - This time letters were in a rational sequence (caps first)!
  - You could add/subtract 32 to go from upper to lower and back
- Still no accents or "foreign" glyphs

ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

The need for multi-language character sets

- Eventually, manufacturers realized that they could maybe sell more software and hardware outside NAmerica if they supported languages other than English
- But how to deal with those "foreign" symbols?
  - While still allowing for interoperability with Latin text?

Multi-language - 2

- Easy! Go back to 8-bit "codepages"!
  - Use the "upper half" of the 8-bit space (another 128 characters beyond ASCII)
- Each computer sold in a "foreign" country would carry a different code point mapping (a "Codepage")
- Again, this system was invented by IBM
  - And done very systematically
  - Each codepage got a numeric designation
  - And a glyph mapping chart in a book

Codepages - continued

- ASCII was always the "lower half" of the codepage
  - So everyone in the world could benefit from using English
- Windows ASCII (again, 8 bits) is cp437
- IBM "international English" is cp850
- Japanese is cp932 (more on this later)
- PRC Simplified Chinese: cp936
- Korean: 949
- Cyrillic: cp1251
- Latin1: cp1252
- etc.

Codepage 850

850 MS-DOS LATIN 1

	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0	32	0 48	@ 64	P 80	` 96	p 112	Ç 128	É 144	á 160	░ 176	└ 192	ð 208	Ó 224	SHY 240
1	! 33	1 49	A 65	Q 81	a 97	q 113	ü 129	æ 145	í 161	▒ 177	┴ 193	Ð 209	ß 225	± 241
2	" 34	2 50	B 66	R 82	b 98	r 114	é 130	Æ 146	ó 162	▓ 178	┬ 194	Ê 210	Ô 226	‗ 242
3	# 35	3 51	C 67	S 83	c 99	s 115	â 131	ô 147	ú 163	│ 179	├ 195	Ë 211	Ò 227	¾ 243
4	$ 36	4 52	D 68	T 84	d 100	t 116	ä 132	ö 148	ñ 164	┤ 180	─ 196	È 212	õ 228	¶ 244
5	% 37	5 53	E 69	U 85	e 101	u 117	à 133	ò 149	Ñ 165	Á 181	┼ 197	ı 213	Õ 229	§ 245
6	& 38	6 54	F 70	V 86	f 102	v 118	å 134	û 150	ª 166	Â 182	ã 198	Í 214	µ 230	÷ 246
7	' 39	7 55	G 71	W 87	g 103	w 119	ç 135	ù 151	º 167	À 183	Ã 199	Î 215	þ 231	¸ 247
8	(40	8 56	H 72	X 88	h 104	x 120	ê 136	ÿ 152	¿ 168	© 184	╚ 200	Ï 216	Þ 232	° 248
9	) 41	9 57	I 73	Y 89	i 105	y 121	ë 137	Ö 153	® 169	╣ 185	╔ 201	┘ 217	Ú 233	¨ 249
A	* 42	: 58	J 74	Z 90	j 106	z 122	è 138	Ü 154	¬ 170	║ 186	╩ 202	┌ 218	Û 234	· 250
B	+ 43	; 59	K 75	[91	k 107	{ 123	ï 139	ø 155	½ 171	╗ 187	╦ 203	█ 219	Ù 235	¹ 251
C	, 44	< 60	L 76	\ 92	l 108	| 124	î 140	£ 156	¼ 172	╝ 188	╠ 204	▄ 220	ý 236	³ 252
D	- 45	= 61	M 77	] 93	m 109	} 125	ì 141	Ø 157	¡ 173	¢ 189	═ 205	¦ 221	Ý 237	² 253
E	. 46	> 62	N 78	^ 94	n 110	~ 126	Ä 142	× 158	« 174	¥ 190	╬ 206	Ì 222	¯ 238	■ 254
F	/ 47	? 63	O 79	_ 95	o 111	⌂ 127	Å 143	ƒ 159	» 175	┐ 191	¤ 207	▀ 223	´ 239	NBSP 255

But.... Japanese?

- What if you can't fit a culture's "character set" into 256 slots?
- Asian languages, for example:
  - Thai: 40+ alphabetic/phonetic symbols
  - Chinese: 5000 common ideographs, 60,000+ total, TWO different systems!
  - Japanese: same (almost) ideographs as Chinese (trad.), PLUS 1 alphabetic series (Katakana), PLUS 1 phonetic (Hiragana), PLUS double-wide latin letters (so they line up with Kanji)!

Japanese - 2

- But a codepage only has an 8-bit namespace!!?
- So, reserve a sequence (or 2, or 3) in the upper half of the table as "pointers" to another 8-bit space
- As in CP932 (Japanese)

Codepage 932

Extract for Lead Byte E9-EA

E9	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	顱 9871	顴 9874	顳 9873	颪 98AA	颯 98AF	颱 98B1	颶 98B6	飄 98C4	飃 98C3	飆 98C6	飩 98E9	飫 98EB	餃 9903	餉 9909	餒 9912	餔 9914
5	餘 9918	餡 9921	餝 991D	餞 991E	餤 9924	餠 9920	餬 992C	餮 992E	餽 993D	餾 993E	饂 9942	饉 9949	饅 9945	饐 9950	饋 994B	饑 9951
6	饒 9952	饌 994C	饕 9955	馗 9997	馘 9998	馥 99A5	馭 99AD	馮 99AE	馼 99BC	駟 99DF	駛 99DB	駝 99DD	駘 99D8	駑 99D1	駭 99ED	駮 99EE
7	駱 99F1	駲 99F2	駻 99FB	駸 99F8	騁 9A01	騏 9A0F	騅 9A05	駢 99E2	騙 9A19	騫 9A2B	騷 9A37	驅 9A45	驂 9A42	驀 9A40	驃 9A43	
8	騾 9A3E	驕 9A55	驍 9A4D	驛 9A5B	驗 9A57	驟 9A5F	驢 9A62	驥 9A65	驤 9A64	驩 9A69	驫 9A6B	驪 9A6A	骭 9AAD	骰 9AB0	骼 9ABC	髀 9AC0
9	髏 9ACF	髑 9AD1	髓 9AD3	體 9AD4	髞 9ADE	髟 9ADF	髢 9AE2	髣 9AE3	髦 9AE6	髯 9AEF	髫 9AEB	髮 9AEE	髴 9AF4	髱 9AF1	髷 9AF7	髻 9AFB
A	鬆 9B06	鬘 9B18	鬚 9B1A	鬟 9B1F	鬢 9B22	鬣 9B23	鬥 9B25	鬧 9B27	鬨 9B28	鬩 9B29	鬪 9B2A	鬮 9B2E	鬯 9B2F	鬲 9B32	魄 9B44	魃 9B43
B	魏 9B4F	魍 9B4D	魎 9B4E	魑 9B51	魘 9B58	魴 9B74	鮓 9B93	鮃 9B83	鮑 9B91	鮖 9B96	鮗 9B97	鮟 9B9F	鮠 9BA0	鮨 9BA8	鮴 9BB4	鯀 9BC0
C	鯊 9BCA	鮹 9BB9	鯆 9BC6	鯏 9BCF	鯑 9BD1	鯒 9BD2	鯣 9BE3	鯢 9BE2	鯤 9BE4	鯔 9BD4	鯡 9BE1	鰺 9C3A	鯲 9BF2	鯱 9BF1	鯰 9BF0	鰕 9C15
D	鰔 9C14	鰉 9C09	鰓 9C13	鰌 9C0C	鰆 9C06	鰈 9C08	鰒 9C12	鰊 9C0A	鰄 9C04	鰮 9C2E	鰛 9C1B	鰥 9C25	鰤 9C24	鰡 9C21	鰰 9C30	鱇 9C47
E	鰲 9C32	鱆 9C46	鰾 9C3E	鱚 9C5A	鱠 9C60	鱧 9C67	鱶 9C76	鱸 9C78	鳧 9CE7	鳬 9CEC	鳰 9CF0	鴉 9D09	鴈 9D08	鳫 9CEB	鴃 9D03	鴆 9D06
F	鴪 9D2A	鴦 9D26	鶯 9DAF	鴣 9D23	鴟 9D1F	鵄 9D44	鴕 9D15	鴒 9D12	鵁 9D41	鴿 9D3F	鴾 9D3E	鵆 9D46	鵈 9D48			

EA	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	鵝 9D5D	鵞 9D5E	鵤 9D64	鵑 9D51	鵐 9D50	鵙 9D59	鵲 9D72	鶉 9D89	鶇 9D87	鶫 9DAB	鵯 9D6F	鵺 9D7A	鶚 9D9A	鶤 9DA4	鶩 9DA9	鶲 9DB2
5	鷄 9DC4	鷁 9DC1	鶻 9DBB	鶸 9DB8	鶺 9DBA	鷆 9DC6	鷏 9DCF	鷂 9DC2	鷙 9DD9	鷓 9DD3	鷸 9DF8	鷦 9DE6	鷭 9DED	鷯 9DEF	鷽 9DFD	鸚 9E1A
6	鸛 9E1B	鸞 9E1E	鹵 9E75	鹹 9E79	鹽 9E7D	麁 9E81	麈 9E88	麋 9E8B	麌 9E8C	麒 9E92	麕 9E95	麑 9E91	麝 9E9D	麥 9EA5	麩 9EA9	麸 9EB8
7	麪 9EAA	麭 9EAD	靡 9761	黌 9ECC	黎 9ECE	黏 9ECF	黐 9ED0	黔 9ED4	黜 9EDC	點 9EDE	黝 9EDD	黠 9EE0	黥 9EE5	黨 9EE8	黯 9EEF	
8	黴 9EF4	黶 9EF6	黷 9EF7	黹 9EF9	黻 9EFB	黼 9EFC	黽 9EFD	鼇 9F07	鼈 9F08	皷 76B7	鼕 9F15	鼡 9F21	鼬 9F2C	鼾 9F3E	齊 9F4A	齒 9F52
9	齔 9F54	齣 9F63	齟 9F5F	齠 9F60	齡 9F61	齦 9F66	齧 9F67	齬 9F6C	齪 9F6A	齷 9F77	齲 9F72	齶 9F76	龕 9F95	龜 9F9C	龠 9FA0	堯 582F
A	槇 69C7	遙 9059	瑤 7464	凜 51DC	熙 7199											
B																
C																

CP 932, continued

- This gives us an effective 16-bit space
  - Almost, sort of
  - It's really a partially-chained space
- But requires that we allow for 2-byte (and later, even 3-byte) sequences to describe one "character"
- You'd better know which CodePage you're dealing with for every string

CP 932, continued

- BUT: still no way to combine multi-CP values in one string
- Unless you start inventing special "opcodes" to switch codepages in the middle of a byte stream
- No one thought that was a good idea
  - So that particular problem was ignored for a while

LMBCS - Lotus MultiByte Character Set

- Lotus invented a new scheme as part of the work done for 123 Release 3
  - The first Lotus spreadsheet coded in C
  - Ran on both DOS and OS/2 (NOT Presentation Manager, which didn't exist yet)
  - Shipped in March, 1989

LMBCS, continued

- Essentially also a "lead-byte" system, taking advantage of the fact that the ASCII codepoints 01 through 1F are non-printing characters
  - 00 is still special, particlearly for C strings
- So, let's assign every glyph to a "code group", every code group has a unique identifier, a value between 1 and 1F
- Each code group corresponds (approximately) to a CodePage

LMBCS, continued

- Code group 1 is Latin1
  - Very close to IBM CP850
- Each code group is either single byte, or multi byte
- Each code group's "lower half" is ASCII
- Code group 16 (Japanese) is multi byte, with designated "lead bytes", very similar to CP932
- And so on...

LMBCS, continued

- The big LMBCS innovation:
  - Every character in "canonical" LMBCS is preceded by its code group identifier

"ABC" in hex format:
01-41-01-42-01-43-0

LMBCS, continued

- Characters in multi-byte code groups could therefore be 2, 3, or 4 bytes long
- 0 still used (in C) as a string terminator
- Guaranteed to be no embedded 0s in a string

Example of 2-byte Japanese character:
10-<lead byte>-<2nd byte>-0

LMBCS, continued

- But this is obviously flawed!
- All strings double in size!
- So we apply an "optimization"
- Each application file comes with a "default code group" designation
- Any character where the code group value is omitted is assumed to be in the default code group
- We detect code group identifiers because they are non-printing characters in a specified range
- Any character in a non-default code group must have its code group value specified

LMBCS, continued

- This is known as "compressed", or "optimized" LMBCS

"ABC" in default code group 01:
41-42-43-0

"ABC" in default code group 16:
01-41-01-42-01-43-0

Hiragana small o in code group 16:
82-A7-0

Hiragana small o in code group 1:
10-82-A7-0

Special Topics

- What about diacriticals?
- What about BIDI (bi-directional) character sets?
- What about "ligatures"?
- How to handle the huge Chinese/Japanese/ Korean character namespaces?
- These are issues that all multi-language systems must handle properly
- (There are even more issues, but these are the common ones)

Diacriticals

- We have a choice for dealing with accent marks
  - Create a code point and glyph for every combination

a á à

e é è

Diacriticals - 2

- Or,
  - Invent the "non-spacing character"
- The "base" character has its own code point, each accent has its own
- Everyone knows not to move the cursor for the non-spacing character

a + ` = à

e + ' = é

Diacriticals - 3

- You might want it either way
- Using non-spacing characters might make sorting easier (accent insensitive sorting)
- Or, might want each code point to be separate (accent sensitive sorting)
- LMBCS does not use non-spacing characters

"BIDI"

- Some character sets are written right-to-left
  - Hebrew
  - Arabic
- But numbers are still left-to-right
- Some character sets are traditionally top-to-bottom, right-to-left
  - Japanese
  - Chinese

BIDI - 2

- Mostly it's an issue for the rendering software
  - And for the input software
- But, how are the code points represented in memory?
  - Which order?
- The convention is that strings are represented in memory in "logical" order
  - I.E., in the order that you input them

BIDI - 3

- So, if you have a RTL sequence of 3 characters
  - Let's represent as: "ZYX"
- Followed by 3 digits ("123")
- You would see those code points in memory as:
  - XYZ123
- The rendering software has to know to reverse the letters, not the digits

Ligatures

- Historically, physical typefaces have combined certain individual letter combinations in to a single glyph
  - ff, ffl, fl, fi, ffi
  - Because the tip of the 'f' overhangs the next character
- Most computer fonts ignore this
- Some don't, they want to render as historically accurate
  - You need to know the character following the 'f' before you can render

Ligatures - 2

- High-end publishing systems want to do this
- So room was made in some code pages for special glyphs representing the ligatures
- Makes sorting very complicated!

Chinese, Japanese, Korean

- Present a special problem, because to fully represent the writing system would consume over 150,000 code points
- Even fully 16-bit systems only give you 65,000 code points
- Compromise: Consolidate
  - A large percentage of the Traditional Chinese, Japanese Kanji and Traditional Korean character sets are held in common
  - So we can represent the set of common glyphs only once
- Referred to as CJK Consolidation

Limitations of LMBCS

- It's a very robust technique for representing multi-lingual text
  - Including within a single string!
  - In a reasonably compressed format (no embedded 0s)
- But...
  - The rest of the world adopted Unicode
- LMBCS remains a "proprietary" Lotus technology

Where is LMBCS?

- LMBCS is used uniformly throughout Lotus products
  - SmartSuite
  - Notes
  - Domino
- And nowhere else
  - Major investment in supporting software tools
  - More on this later

What Does Everyone Use Now?

- Unicode
  - Lotus also supports Unicode, as we'll see
- Unicode is an open standard
  - See http://www.unicode.org

What is Unicode?

- Earlier character encoding schemes are often referred to as SBCS or DBCS or MBCS
  - Single Byte Character Set (e.g., Ascii)
  - Double Byte Character Set (e.g., CP932)
  - Multi-Byte Character Set
- Unicode is none of these

What is Unicode?

- Unicode is not byte oriented, so calling it even MBCS is misleading
  - Though most of the time this is a true statement
  - There are times when you can treat it as an SBCS too
- My working definition:
  - A character encoding scheme that assigns a unique numeric value to every character
  - All written characters for all human languages (and many special characters as well) are accommodated

What is Unicode?

- Note: this definition says nothing about how big the values might be
  - Or how many "bytes" might be needed to represent any given character
  - Because that's not the important issue
- The key point is that the "name space" for characters in Unicode is flat
  - 0 is still a special value
  - Upper value is essentially unbounded
  - And therefore very extensible

What is Unicode?

- The current standard defines an upper limit of 0x10FFFF (using 3 8-bit bytes)
  - More than one million code points
  - Of which about 5% is allocated
  - 13% is reserved for private use
  - 2% reserved
  - 5% planning underway
  - Plenty of room for growth
- Of course, the devil is in the details

Unicode Mappings

- The very large Unicode character space is laid out in regions
- Designed for interoperability with Ascii and ISO Latin-1
  - For ease of translation
- Ascii values (up to 0xff) are maintained unchanged
- Standards committee determines the rest of the values

Unicode Encodings

- Question on computer representations:
  - 0x10ffff is 3 bytes, 24 bits
  - Most computers do not align on 3-byte boundaries
  - So we would "round up" to 4 bytes, 32 bits
- So, do we need to allocate 32 bits per character in all our strings?
  - And, if we do, won't there be embedded 0s?

Encodings - 2

- Yes, there would
- But we don't necessarily need all 32 bits for each character.
- There are alternate Unicode "encodings":
  - UTF-8: each char is 1, 2, 3, or 4 units
  - UTF-16: each char is 1 or 2 units
  - UTF-32: each char is 1 unit
- UTF-16 is the "default" encoding

Encodings - 3

- A char's real value does not change with the encoding
  - Only the number of bytes used to represent that value

"A"	in UTF-8:	0x41	(byte)
	in UTF-16:	0x0041	(short)
	in UTF-32:	0x00000041	(DWORD)

Encodings - 4

- Implication:
  - In UTF-8 and (to a lesser extent) in UTF-16
  - You may STILL need multiple units to represent a given code point
  - Therefore, Unicode can still be "multi-byte", in a sense
  - Meaning, "multiple positions"
- Zero still used by convention to terminate strings
  - Though this can vary by programming language
  - And the number of bytes must conform to the encoding (UTF-8, 16, 32)

A Point on Diacriticals

- Unicode also gives you a choice on representation
  - One glyph for a "pre-composed" character
    - U-umlaut is 0x00fc
  - Base char + non-spacing char ("composed")
    - U + umlaut is 0x0075 + 0x0308
- So even in UTF-32 you may need 2 positions

Referencing Characters in Unicode

- All characters have a unique name
  - Based on ISO 10646
  - E.g., "Bengali Digit 5"
- The convention for representing a character's code point value in Unicode
  - U+xxxx
  - Always assumes Hex representation

Software for multi-language character sets

- The "standard" C library string functions are no longer usable
  - They all assume 1 byte == 1 char
  - strlen() tells you the length of a string in bytes, but not how many chars are in it
  - strcmp() is only valid for *some* SBCS, would never work with (e.g.) EBCDIC
  - Cannot add 0x20 to go from uc to lc
- You *must* use a software library that supports the character set in use

Software for multi-language character sets - 2

- What kinds of things do you need to do?
  - Length of string (chars, bytes)
  - Search for char in string
  - Search for substring in string
  - Move cursor within string (by char):
    - First, Last
    - Next, Prev
  - Truncate at a certain byte position

Software for multi-language character sets - 3

- Tasks, continued:
  - Test type of char (digit, alpha, upper, lower, accented, not accented...)
  - Modify casing (upper, lower)
  - Modify accenting (with, without)
  - Sort (compare 2 strings lexically)
  - Translate to/from native character set

A Word on Sorting

- All lexical sorting requires is the ability to compare any 2 characters and say one is < the other
- BUT: it's a complex topic
- Easy sorting simply compares raw code point values
  - 'A' < 'a'
  - 'a' < 'b'
  - '1' < '9'
- But where are numbers relative to letters?
  - In Ascii, all digits are < all letters
  - But not universally true!

Sorting - 2

- Need to handle comparison of characters using more than one byte
- Need to handle composed characters, too
- Real-world products need to offer multiple compare/sort options:
  - Case sensitive/insensitive
  - Accent sensitive/insensitive
  - Width sensitive/insensitive (for Asian double-wide characters)
  - Number first/numbers last

Software for multi-language character sets - 4

- Topics:
  - Unicode and Java
  - MLCS and Fonts: Rendering text
  - LMBCS and Unicode in Notes

Unicode and Java

- Java is all Unicode internally
  - Generally UTF-16 encoding
  - "char" is a 16-bit quantity in most cases

Editing Java Code

- Be careful using non-Ascii characters as literals in Java code
  - 'a' is fine
  - Special characters ('\t', '\n') are ok
  - 'KK' (some Kanji character) is dangerous
- This is editor-dependent
  - If your text editor supports Unicode, then it's ok
  - Otherwise, your code may not be compiled on all systems

Editing Java Code - 2

- Not an issue for Java code entered directly in Domino Designer
  - It's a LMBCS editor
  - Source code is stored in UTF-16
- This is not an issue for LotusScript or @Function code
  - Because you are always using a LMBCS editor

Unicode and Java - 2

- Java lets you specify the code point value for a single char:
  - '\uXXXX'
  - Must supply 4 digits, implying UTF-16
- byte is an 8-bit quantity in Java
  - Be careful! Smaller than a char!
- int is a 32-bit quantity in Java
  - So cannot always freely convert
  - c = (char) i; works *most of the time*
  - i = (int) c; always works

Unicode and Java - 3

- Character manipulation is built-in to String/StringBuffer classes
  - char c = string.charAt(i)
  - toUpper/LowerCase()
  - toCharArray()
  - search, replace, etc., etc.
  - append()

Java Type Conversion

- Java will convert between String and char[] transparently
- Java will convert between String and byte[] with a specified encoding

```
byte[] arrOfBytes = null;
String st = "Some string";
arrOfBytes = st.getBytes("UTF-8");

String st2 = new String(arrOfBytes, "UTF-8");
```

Java Type Conversion - 2

- Java will convert between String and char[] transparently
- Java will convert between String and byte[] with a specified encoding

```
byte[] arrOfBytes = null;
String st = "Some string";
arrOfBytes = st.getBytes("UTF-8");

String st2 = new String(arrOfBytes, "UTF-8");
```

NOTE: length of st2 (chars) may not = length of arr (bytes)!

Java Type Conversion - 3

- Useful for converting byte-oriented strings (e.g., from C programs) to "real" ML strings
- Example: Notes converts all strings to UTF-8 when passing them to Java
  - But what if it's Kanji?

```
String utf8 = document.getItemValueString("abc");
byte[] bytes = utf8.getBytes("UTF-16");
String kanji = new String(bytes, "UTF-16");
```

Java Supports Localization

- java.util.Locale
- Represents:
  - A "language code", e.g., "en", "de"
    - ISO 639
  - A "country code", e.g., "US", "CA"
    - ISO 3166
  - (Optional) A "variant", e.g., "posix", "MAC"
    - vendor specific

Localization - 2

- A Locale instance implies something about the usage/formatting of:
  - dates (9/10/01 vs. 10/09/01 vs. 10.09.01)
  - currency symbols
  - accented characters
- So, these are all different:
  - en/US, en/UK, en/IR
  - fr/CA, fr/FR, fr/BE

Localization - 3

- But: Locale is independent of display (font)
- Can query Java for the current, "default" locale
  - Dependent on system configuration
- Locale instances can be used to auto-format strings for display

Java, LotusScript for File I/O

- If the file system is Unicode (e.g., Windows NT), then there are no issues
- Otherwise, you have 2 choices:
  - Write binary data in Unicode format
    - Not readable by other system tools
  - Convert going in and out
- LotusScript does Choice 2 automatically

Java and File I/O

- With Java, you can control how you do it
- Package java.io.*
  - Base layer consists of InputStream and OutputStream
  - Byte oriented, you figure it out
  - Other layers support richer data types

Java and File I/O

- What's wrong with this code?
  - Works, *sometimes...*

```
FileInputStream fis = new FileInputStream("d:\temp\abc.txt");
char c; int i;
while ((i = fis.read()) >= 0)
{
    c = (char) i;
    // etc....
}
fis.close();
```

Java and File I/O

- It works when:
  - File contents are Ascii
  - File contents are Unicode
- Otherwise, you can't assume that raw bytes are cast-able to Unicode
  - Correctly, that is
  - You are likely to get garbage

Java and File I/O

- For "native" file i/o, use Reader/Writer classes
  - char oriented, not byte oriented

```
FileReader fr = new FileReader("d:\temp\abc.txt");
char c; int i;
while ((i = fis.read()) >= 0)
{
    c = (char) i;
    // etc....
}
fr.close();
```

MLCS and Fonts: Rendering Text

- Rendering means:
  - Displaying a stream of code point values on a screen, or on paper
- Requires mapping a code point value to a particular glyph
  - Or a sequence of code point values to a single glyph
- Requires a displayable glyph for the code point value
  - With attributes too (bold, italic, superscript, etc.)

Rendering - 2

- A "Font" (in computer terms) is a set of glyphs
  - Generally a table of displayable glyphs
  - Upper and lower case letters, digits, etc.
  - Kanji characters
- Can have multiple Fonts for any language character set
  - Just a different way of rendering each glyph
  - Times New Roman vs. Helvetica

Rendering - 3

- Not all fonts have glyphs for the entire Unicode space
  - Most don't
- "Missing" glyphs default to a "fallback" character
  - The character data is still all there!
  - It's only the rendering software that can't handle the "missing" glyph

Rendering - 4

- The character set that a font maps to is vendor-dependent
- Some are Ascii only
- Some are Asian only
  - Though most Asian fonts also include Ascii
- Others handle (some subset of) Unicode
- Rendering software is responsible for directionality
  - BIDI
  - Top-down

Rendering - 5

- The amount of space a given string uses on the screen (or on paper) depends on:
  - Font characteristics (monospaced, proportional)
  - Attributes (bold, plain)
  - Point size (10, 12, 48...)
  - Pixel resolution of the display (DPI)
  - Actual chars used (esp. if proportional)
- Can be very complex to compute!

Java & LMBCS in Notes

- All system display software requires "native" character set
- All Lotus product code bases use LMBCS as the character set
  - Internal manipulation (sorting, searching, etc.)
  - Persistent storage (on disk)
- With one exception:
  - LotusScript stores all strings as Unicode (UTF-16)
  - Must convert between:
    - LMBCS (for Notes)
    - Native (for display)

Calling External Code From Java & LotusScript

- LotusScript calls externally using "Declare" statements
  - Describes library where code resides, name of entry point, and arguments
  - For String types, can declare the arg as:
    - "ByVal stringarg As *LMBCS* String"
    - Does conversion automatically
    - MUST be "ByVal"
- Otherwise, String is Unicode

Calling External Code From Java & LotusScript - 2

- In Java, you can't invoke C entry points directly
  - Must use Java Native Interface (JNI)
  - C entry point receives a pointer to a Java services vector, plus arguments
    - Args are scalars or Java object handles
- You use the JVM services vector to manipulate the arguments
  - E.g., convert String objects to UTF-8 string buffers

UI Support for Composed Characters

- In the Notes Client you can use Alt-F1 to compose chars
  - Alt-F1 + a + ` = à
  - Alt-F1 + c + , = ç
  - and so on

Notes - 2

- Most conversions are transparent to the LS developer
  - Java Strings converted to Notes LMBCS
  - Notes LMBCS strings converted to Unicode (UTF-8)
  - LS strings converted to Notes LMBCS
  - Notes LMBCS converted to Unicode (UTF-16)

Notes - 3

- When using Notes C API from LotusScript, need to worry about string type
  - Declare .... (...., st byval As LMBCS string)
  - Converts automatically to LMBCS
- When using custom C DLLs called from LotusScript
  - Either do same Declare trick
  - Or receive strings as UTF-16 Unicode

Notes - 4

- For Java, when calling C:
  - String objects received as object handles
  - Use Java Native Interface (JNI) services to extract string value the way you want it
    - UTF-8, UTF-16
- Notes has many entry points in the C API for dealing with multi-language strings

Multi-language APIs

- Categories:
  - Translation (OSTranslate, NLS_Translate)
  - Parsing, searching (NLS_XXX)
- All calls are documented in the Notes C API toolkit
- NOTE: All "char *" decls in the C API mean "unsigned char"
  - LMBCS

OSTranslate

- The key is the "TranslateMode" options
  - Note: result-length may vary from input-length!

```
WORD LNPUBLIC OSTranslate(
    WORD  TranslateMode,
    char far *In,
    WORD  InLength,
    char far *Out,
    WORD  OutLength);
```

OSTranslate - 2

- Assumes that either source or destination string (or both) is LMBCS Group 1
  - Notes assumes Group 1 strings are optimized
    - Leading group indicator-bytes compressed out
  - All others non-optimized

OSTranslate - 3

OS_TRANSLATE_NATIVE_TO_LMBCS - Convert input string from machine's native character set to LMBCS

OS_TRANSLATE_LMBCS_TO_NATIVE - Convert input string from LMBCS to machine's native character set.

OS_TRANSLATE_LOWER_TO_UPPER - Current international case table.

OS_TRANSLATE_UPPER_TO_LOWER - Current international case table.

OS_TRANSLATE_UNACCENT - International unaccenting table. NO REVERSAL!!

OS_TRANSLATE_OSNATIVE_TO_LMBCS - Same as NATIVE (not documented!)

OS_TRANSLATE_LMBCS_TO_OSNATIVE - Same as TO_NATIVE (not documented!)

OS_TRANSLATE_LMBCS_TO_ASCII - Convert the input string from LMBCS to character text.

OS_TRANSLATE_LMBCS_TO_UNICODE - Convert the input string from LMBCS to UNICODE.

OS_TRANSLATE_LMBCS_TO_UTF8 - Convert the input string from LMBCS to UTF8.

OS_TRANSLATE_UNICODE_TO_LMBCS - Convert the input string from UNICODE to LMBCS.

OS_TRANSLATE_UTF8_TO_LMBCS - Convert the input string from UTF8 to LMBCS.

OSTranslate - 4

- Note that multiple translations are "lossless"
  - If you go one way, then come back, all data is preserved
- EXCEPT:
  - Accented to unaccented
  - LMBCS to Ascii (maybe)
- Complicates display input and editing

National Language System (NLS) APIs

- Supports low-level character based parsing, searching and translation
- Any character set, not just LMBCS/1 or current native

NLS_PINFO

- Use NLS_load_charset to read a charset descriptor table from disk
- Returns a NLS_PINFO *
  - Opaque data structure, but you will need it
- List of character set IDs found in NLS.H
- Used in NLS_translate

NLS-translate

- More functionality than in OSTranslate
  - Arbitrary translation to/from any character set
  - NLS_translate

```
NLS_STATUS LNPUBLIC NLS_translate(
    BYTE far *pString,
    WORD  Len,
    BYTE far *pStringTarget,
    WORD far *pSize,
    WORD  ControlFlags,
    NLS_PINFO  pInfo);
```

NLS Translation Options

NLS_NONULLTERMINATE - Does not add a NULL to the end of the translated result.

NLS_NULLTERMINATE - Adds a NULL to the end of the translated result.

NLS_STRIPUNKNOWN - Strips unknown characters.

NLS_TARGETISLMBCS - Converts target string to LMBCS.

NLS_SOURCEISLMBCS - Converts source string from LMBCS.

NLS_TARGETISUNICODE - Converts target string to UNICODE.

NLS_SOURCEISUNICODE - Converts source string from UNICODE.

NLS_TARGETISPLATFORM - Converts target string to the encoding used by the host OS.

NLS_SOURCEISPLATFORM - Converts source string from the encoding used by the host OS.

Other NLS API Calls

- NLS_string_chars
  - Number of chars in a string
- NLS_string_bytes
  - Number of bytes in a string (same as strlen)
- Test a character for type:
  - isdigit, isupper, islower, isspace, ispunct, iscntrl, isalnum, isalpha, isarith,

Other NLS API Calls - 2

- isleadbyte
  - If True, then the next 1 (or 2, or 3) bytes are part of the same character
- Enumerate through a string
  - NLS_get
- Parse strings
  - goto_next_break
  - goto_next/prev_word_end/start

Other NLS API Calls - 3

- Search
  - find, find_substr

String Truncation

- String truncation: don't ignore this!
  - Failure to do this properly can cause crashes
- Situation:
  - You have an N-byte long MLCS string
  - You must truncate to M bytes
    - $M < N$
    - For display, to fit in a fixed buffer, whatever

String Truncation - 2

- Can't just chop it off at M bytes!
  - Might be the middle of a multi-byte sequence
  - Will result (at least) in garbage on the display
- Correct technique:
  - Get pointer to max slot in string
  - Call NLS_goto_prev
    - Pass in ** for current position, * for start of string
  - Moves your pointer to the start of the previous char
  - Can truncate there, or at start of next char

MLCS APIs - Summary

- All the old C techniques are now useless
  - Worse: wrong!
- You will need to accustom yourself to a new set of assumptions and functions
- But once you get comfortable with it, it's not difficult
  - Make it habitual!
- NLS calls somewhat limited:
  - Only a few of the string-compare and search equivalents for MLCS are exposed!
  - Testers for full/half pitch not exposed!

Web applications and MLCS

- What happens Notes data served to the Web?
- HTTP allows specification of the page charset
  - Client can specify desired charsets on request
  - Server specifies charset on result page

Web Applications - 2

- Notes defaults to ISO Latin-1 for output
  - Automatically converts from LMBCS
- But, how do you know what the Client's real charset is?
  - Look at requested charsets
  - Look at country/language code
- Can you tell Domino what charset to use on output?

Web Applications - 3

- Yes!
- Domino examines Web agent output for HTTP headers
  - Must occur BEFORE any data
  - Domino will notice them and pass them through
  - If you specify a Content-Type header, Domino will try to translate

Web Applications - 4

- Just print out the header (in the correct format) as the first thing:

```
print "Content-Type: text/html; charset=ISO-8859-4
print ""      ' null to force a newline: REQUIRED
print "Yo, world!"
```

Web Applications - 5

- Unfortunately, only works from agents
- No way (that I know of) to build this into a form or page

Web applications and MLCS

- Another trick for using in browsers
- HTTP specifies that Unicode values can be substituted for any char
  - 'Space' == U+0020
  - Can encode for HTTP as
    - %0020
    - Or %20
- So you can encode arbitrary Unicode chars in your Notes form or page!
  - The browser will interpret it correctly
  - Very useful when encoding pass-thru URLs

Summary

- If everyone spoke (and wrote!) only one language, it would be easy
- Remember: character set encodings are independent of rendering
  - Though typographical legacies complicate encodings (e.g., ligatures)
- Unicode is an important technological advance
  - Everything defined in a single "space"

Summary - 2

- But the tools are not yet sophisticated enough
  - Developers still need some knowledge of how character sets work
  - And Unicode implementations are not always uniform
    - UTF-8? 16? 32?
    - Java UTF-16 still has multi-char code points

Summary - 3

- It will keep getting better
  - A good thing
- It will keep changing
  - Not always so good

Click here to type page title

Q & A