Asyntactic Script

and the **Base16b** encoding family

an Open Specification

by Andrew Henderson

Base16b.org

Table of Contents

Asyntactic Script....1
- Abstract....4
  - Author....4
  - Version....4
  - Licensing....4
  - Open Specification....4
  - Originality....4
- Unicode....5
  - Scripts in Unicode....5
    - Script examples....5
  - Character syntax....5
    - Semantics....5
- Asyntactic Script....6
  - Definition....6
  - Intended purpose....6
    - Script as substrate....6
    - Code Points and values....6
    - No special characters....6
  - Private Use....6
    - Authority....7
    - Co-existence with other private scripts....7
  - Nomenclature....7
    - Appellation....7
- Encoding / decoding....8
  - Family of bases....8
    - Subsumption of bases....8
    - Equivalence of Asyntactic script and Base17b range....8
  - Mapping....8
    - Source data preparation....8
    - Conversion by mapping table....8
      - Mapping table contiguity....8
      - First and Last characters....9
      - Base16b....9
      - Base17b....9
      - Base7b to Base15b....10
  - Termination character....11
    - Dual Purpose....11
    - Encoding the termination character....11
      - Working base....11
      - Remainder....11
    - Inversion....11
      - Encoding example....11

Decoding the Working Base....12
Decoding the Remainder....12
Variable length characters....12
Readability....13
Font....13
Fall-back font....13
Not for your eyes....13
Performance....14
UTF-8....14
Information per byte....14
Information per character....14
Other Unicode variants....14
Expected / actual....15
Caveats....15
Side by side comparison....15
Signalling use of the Asyntactic script....15
Applications....15
Micro-blogging....15
Non-standard syntax....15
Caution....16
Unicode support....16
Improper implementations....16
Missing font....16
Conclusion....16
Appendix....17
Performance tables....17
Sequence of bases....18
Experiment....19
Results....19
Visual comparison....19
Code sample....20

Abstract

A new script in the Private User Areas[1] (PUA) of the Unicode address space is specified.

The script's intended purpose is to facilitate encoding of binary data using large sets of the script's characters. This paper describes how to use the Asyntactic script to encode and decode data in a family of bases consisting of characters ranging in number between 2^7 and 2^{17}.

Efficiency of the encoding, relative to existing methods is recorded. The top performing member of this family encodes information per character at nearly three times greater density than Base64[2] encoding.

Author

This paper was written by Andrew Henderson of Base16b.org.

If you would like to contact the author please use the contact details published at the web site[3].

Version

This is the initial Draft 0.1 of this document.

Licensing

There is no charge for use of this document, which is published under a Creative Commons License[4]

The appended code sample20 may be used and distributed according to the MIT license.

Open Specification

The Asyntactic script is an open specification.

If and when its usage becomes widespread, the specification of Asyntactic script and associated $Base_{16b}$ family of encodings will be submitted to IETF[5] for inclusion as as an Internet Standard. $Base_{16b}$ might likewise be submitted for Mime Type recognition under the IANA rules.

Originality

This work is the author's own creation. As far as the author is aware, no Unicode script having properties like the Asyntactic script has been proposed. Neither is he aware of any extant or proposed binary encoding scheme in bases far above base64, like those proposed in this paper.

1 http://en.wikipedia.org/wiki/Private_Use_Area#Private_use_characters
2 http://tools.ietf.org/html/rfc4648
3 http://www.base16b.org/library/js/base16b.js
4 http://creativecommons.org/licenses/by-nd/3.0/
5 http://ietf.org

Scripts in Unicode

In Unicode, a script defines a range of characters having common function, coherent syntactic representations and, typically, contiguous locations in the code point address space of the complete Unicode specification.
According to the code point range to which it is assigned a script is either a formal or an informal part of the Unicode specification. Informal scripts must be in one or more of the Private User Areas (PUA). Scripts in the PUA are not subject to control by the Unicode authority.

Script examples

The table provides some examples of Unicode scripts, with a sample character from each. The Unicode official website[6] has a comprehensive list of the formally approved scripts.

	Plane			Range (Hex.)		Qty	sample character		
	#	Type	Script	From	To	(Dec.)	Code	Representation	English Semantics
Formal	0	BMP	Controls + Basic Latin	0000	007F	127	0025	%	per one hundred
Formal	0	BMP	Han	4E00	9FFF	20,991	6D77	海	the sea
Informal	0	PUA	Klingon	F8D0	F8FF	47	F8D2	(see below)	“ch”
Formal	1	SMP	Domino	1F030	1F093	99	1F093	(see below)	double six (vertical)
Informal		PUA	Asyntactic	(see below)			10888D	<any>	<none>

Table 1: Examples of Unicode scripts and sample characters

(Follow these links for graphical syntactic representations of sample characters in the Klingon[7] and Domino[8] scripts).

Character syntax

For the purposes of this paper, syntax of a Unicode character is defined as a standard or conventional graphical representation (glyph) of a Code Point[9]. Whether specified by the Unicode authority[10] or by some private organisation. The definition need not be rigorously specified. It is the absence of an identifiable character that is significant.

Semantics

Unicode does not concern itself with the semantics of characters. However, explicit or conventional semantics of the characters in a script are typically documented by organisations for whom the script is significant. Character semantics may be defined formally, by convention, contended by rival organisations or not described at all. In different contexts, more than one interpretation may be ascribed to the same script character. A good example of an organisation which formally specifies the semantics of its component characters is MathML[11].

6 http://www.unicode.org/charts/
7 http://www.kli.org/tlh/pIqaD.html
8 http://www.unicode.org/charts/PDF/U1F030.pdf
9 http://www.unicode.org/versions/Unicode5.0.0/ch03.pdf#G2212
10 http://www.unicode.org/consortium/consort.html
11 http://www.w3.org/TR/2008/WD-xml-entity-names-20080721/

Asyntactic Script

Definition

The Asyntactic script is a hypothetical Unicode script whose definition, as implied by its name, explicitly specifies what the script lacks. The Asyntactic script has the following attributes :

- There is no syntactic representation of a character in the Asyntactic script that can be considered standard or conventional. This is roughly equivalent to there being no Abstract Characters[12].
- Character representations, wherever used are arbitrary glyphs which can be freely substituted without loss of information.
- Any character or set of characters encoded in Asyntactic script must always be assumed to have acquired no semantic value by virtue of its syntactic representation.

Intended purpose

Comparable to Base64, the intended purpose of the Asyntactic script is the efficient encoding of arbitrary binary data into a text format. Data encoded in the Asyntactic script may then be distributed through media that support some variant of Unicode such as UTF-8[13] or UTF-16[14], but where intermediate transport layers are not necessarily 8-bit clean.
Unintended uses are also possible. Developers are not discouraged from using the Asyntactic script for alternative purposes.

Script as substrate

The Asyntactic script can be thought of as the substrate for base encoding. The script needs to be syntax-free because it is not feasible to manage the Abstract Character of thousands of characters.

Code Points and values

Whether used for binary data encoding or for some other purpose, the significance of a character in the Asyntactic script always lies exclusively in the numeric value implied by the character's code point to value mapping. After translating a character's code point using appropriate the mapping table, the mapped value may be passed into consuming applications as hexadecimal, binary or some other suitable raw data format.

No special characters

There is nothing special about any code point in the Asyntactic script. Each is simply and exclusively a mapping to a numeric value under a specific working base.

Private Use

Planes 15 and 16 and the relevant section of Plane 0 are specified for Private use. It is appropriate that the Asyntactic script should continue to operate in these code points. There will be no need to eventually try to 'upgrade' to one of the Unicode public spaces.

12 http://www.unicode.org/versions/Unicode5.0.0/ch03.pdf#G2212
13 http://unicode.org/resources/utf8.html
14 http://unicode.org/standard/principles.html

Authority

The Asyntactic script is entirely in Unicode's private use areas (PUA).

Beyond specifying the script according to the specification of Unicode, there is neither procedural nor practical reason to seek approval or consensus for the specification or use of the Asyntactic script. Neither from the Unicode governing authority nor from other private organisations who may be using the same code point range for some other purpose.

Co-existence with other private scripts

In cases where the Asyntactic script shares code point space with other Private Use script(s) in the same environment, the syntactic representation of (some of) the Asyntactic scripts characters may be determined by those other scripts. Visually, some characters of $Base_{16b}$ encoded script might appear as, say an Apple hack[15] glyph, if such a font has been installed on the device.
Since the Asyntactic script has no native character set by definition, such overlap cannot be a burden for legitimate uses of the Asyntactic script, nor for applications using those other private scripts.

Nomenclature

If we had continued the tacit naming convention of Base 16 (Hex), Base 32 and Base 64, $Base_{16b}$ might have been called Base 65536. That would be unwieldy, so a new naming convention is introduced18.

The word Base (capitalised or not) is followed by a subscripted integer in the range 7 -17 then the subscripted letter b (standing for bits).

This naming documents that the Asyntactic script style of base encoding is described. The number before the b is the number of bits per character in the encoding method. The largest is $Base_{17b.}$

Though not members of the Asyntactic script base encoding family, Binary, Base 4, Octal, Hex, Base 32 and Base 64 are shown in the Appendix18 in order to illustrate how base encodings in the Asyntactic script are a natural extension of those more familiar bases.

Appellation

The common appellation of this family of encoding is $Base_{16b.}$ For most purposes the family should be thought of and described as a single $Base_{16b}$ encoding method. For instance, it would usually be appropriate to describe a $Base_{17b}$ encoded string as being $Base_{16b}$ encoded.

Since sufficiently detailed information is provided in the termination character of every encoded string, working base information does not need to be explicitly documented through meta data.

However, when differentiation from other encoding bases in the family is intended (as necessarily occurs frequently in this document), then the more detailed term should be used. i.e. “technically, this string is $Base_{17b}$, not $Base_{16b}$ encoded”.

15 http://www.evertype.com/standards/csur/conscript-table.html

Encoding / decoding

Family of bases

Ranging from $Base_{7b}$ to $Base_{17b}$, there are 11 variants (family members) of encoding bases present in the Asyntactic script.

One such, $Base_{16b,}$ encodes exactly 65,536 (2^{16}) bits or 64Kib per character. From a $\log_2$ point of view 64Kib is a remarkably round number :

(((2 bits power 2) power 2) power 2) power 2

This roundness is likely to have some significance for practical applications. For instance, $Base_{16b}$ is guaranteed to encode exactly two bytes of an input string into each encoded character. For this reason $Base_{16b}$ is the titular head of this family. Despite that $Base_{17b}$ is the most senior family member.

Subsumption of bases

Every base in the family subsumes every base of a lower order.

$Base_{7b}$ is subsumed by $Base_{8b}$ ... is subsumed by $Base_{17b}$

Code Point : Value mapping of higher order bases includes the exact mapping of all lower order bases.

This implies that, for full characters, it is not necessary to specify a working base in order map code points to values and vice versa. Nonetheless, working base is a necessary input to the algorithm that decodes the value of the remainder from the termination character11.

Equivalence of Asyntactic script and $Base_{17b}$ range

The code point range of $Base_{17b}$, the largest base in the family, is exactly the code point range of the Asyntactic script itself. Their mappings are equivalent.

Mapping

Source data preparation

Binary input data are first prepared for encoding by splitting into segments of b bits in length, where b is the number of bits in a character for the working base.

Conversion by mapping table

Binary data segments are encoded into the Asyntactic script by reference to the code point : value mapping table for the working base.
Data are decoded using the same mapping table in reverse. The mapping tables for the entire range of bases (7b -17b) are specified in the following sections.

Mapping table contiguity

These mapping tables specify the first and last code point characters and values. If not otherwise

specified, the range is contiguous and the order is sequential. So intermediate values can readily be calculated. Sequential, contiguous flow is represented by ellipsis (…)

Where the contiguity of the base's range is broken, the table specifies the exact mappings to be used and then re-establishes sequential contiguity.

First and Last characters

The first and last characters / values in the mapping are indicated in the table. These values will be referenced by the specification of the termination character encoding and decoding algorithm.

$Base_{16b}$

This section documents in detail how $Base_{16b}$ works. Later sections specify how other members of the Asyntactic script base encoding family differ from $Base_{16b}$.

$Base_{16b}$ occupies the entire available range of characters in Unicode's Plane 15 plus a pair of available characters in the private user area of Plane 0.

$Base_{16b}$		Plane 15			Plane 0	
	Code Point	U+F0000	…	U+FFFFD	U+F80A	U+F80B
		first		…		last
Numeric Values	Hex	0x0000	…	0xFFFD	0xFFFE	0xFFFF
	Decimal	0	…	65533	65534	65535

Table 2: Base16b code point : value mapping

There is nothing special about any code point in the Asyntactic script. For instance, $Base_{16b}$'s two extra characters in Plane 0 are just two ordinary characters. They are included to make up the required complement of 65,536 bits (64Kib). Unfortunately Unicode specifies that the final two characters in each plane are not for use, so a contiguous 64Kib code point space was not available.

$Base_{17b}$

There are sufficient code points in the Private User Areas of Unicode 5.1 for one extension: $Base_{17b}$, a natural extension to $Base_{16b}$.

$Base_{17b}$		Plane 15			Plane 0		Plane 16			Plane 0	
	Code Point	U+F0000	…	U+FFFFD	U+F80A	U+F80B	U+100000	…	U+10FFFD	U+F80C	U+F80D
		first		…							last
Numeric Values	Hex	0x0000	…	0xFFFD	0xFFFE	0xFFFF	0x10000	…	0x1FFFD	0x1FFFE	0x1FFFF
	Decimal	0	…	65533	65534	65535	65536	…	131069	131070	131071

Table 3: Base17b code point : value mapping

The specification of $Base_{17b}$ subsumes $Base_{16b}$ and all other lower order bases. It comprises the entire code point spaces of $Base_{16b}$ and the additional code point spaces in Plane 16 and Plane 0 specified the table.

$Base_{7b}$ to $Base15_{b}$

Smaller bases may be used to encode data using characters having numbers of bits between 7 and 15. These bases all fit into a single Unicode plane.

	Plane 15				
	Code Point			Dec. Value	
	first	…	last	first	last
$Base_{7b}$	U+F0000	…	U+F007F	0	127
$Base_{8b}$	U+F0000	…	U+F00FF	0	255
$Base_{9b}$	U+F0000	…	U+F01FF	0	511
$Base_{10b}$	U+F0000	…	U+F03FF	0	1023
$Base_{11b}$	U+F0000	…	U+F07FF	0	2047
$Base_{12b}$	U+F0000	…	U+F0FFF	0	4095
$Base_{13b}$	U+F0000	…	U+F1FFF	0	8191
$Base_{14b}$	U+F0000	…	U+F3FFF	0	16383
$Base_{15b}$	U+F0000	…	U+F7FFF	0	32767

Table 4: Base7b to Base15b code point : value mapping

Termination character

Dual Purpose

The termination character is an efficient way to encode both the value of the remainder and the working base into a single character. As implied by the name, it is the final character in any $Base_{16b}$ family encoded string. Use of a termination character is mandatory.

Encoding the termination character

Working base

So that decoding applications will be able to determine which base is specified as the encoding base and can decode the (remainder portion of) the encoded string appropriately, the value of the working base must be encoded into the termination character. The value of working base must be in the range 7 to 17.

Selection of working base is at the discretion of the application doing the encoding.

Remainder

The input string's bit length is not always divisible exactly by the number of bits in the working Base's character set. There may be some partial character number of bits left over which we call the remainder.

The length of the remainder cannot be the working base's character length. e.g. 13 bits of remainder in $Base_{13b}$ is impossible; that would not be a remainder, it would be a full character. If there were no further input bits after such a full character then the remainder value encoded by the termination character would be zero (0x00).

The termination character, which must be present, must encode a remainder in the range 0 to $2^{(wb-1)}$ -1,where *wb* is the working base.

Inversion

The value implied by the code point of the termination character is the two's complement inversion of the value of the remainder's bits, in the working base.

Encoding example

Having completed the encoding to $Base7_b$ of all the full characters in a binary string, we want to finish by encoding the termination character. There happens to be three bits left over after the last fully encoded character. Those three bits' values are 100 (0x4).

In any $Base16_b$-family such as $Base7_b$, 0x4 would normally be encoded at code point U+F0004. However, because this is the termination character it is encoded at the inverse code point. The inverse code point can be calculated by counting down from the last code point of the working base.

U+F007F ($Base7_b$ last) – 0x4 = U+F7007B

So in this example we'd use code point U+F7007B as the termination character.

Decoding the Working Base

The working base signifies which base an Asyntactic script string is encoded in. Its value lies in the range 7 to 17. The value of the working base is itself encoded in the termination character. To decode the working base's value, simply count the number of bits in the binary value implied by the termination character's code point, ignoring leading zeros.

e.g. termination character at code point U+F01A9 maps to a value of hex 0x1A9 (binary 110101001).

The length of the termination character's binary value is nine, so the working base of this string is 9, signifying $Base_{9b}$ encoding.

This method works because the remainder is never the full length of the base. At least the first bit of the remainder's expanded value must be a leading zero. After inversion this leading zero bit becomes a one, so the encoded termination character's value always becomes the same bit length as the base.

Decoding the Remainder

The value of the remainder can be decoded by inverting the value of the termination character.

e.g. using the above method a string with termination character U+F7F0A is determined to be encoded in $Base_{15b.}$

The encoded string's remainder is calculated by counting down from the last code point in $Base_{15b}$

U+F7FFF - U+F7F0A = (hex) 0xF5 = (bin) 11110101

In this example, eight bits remainder were encoded in the termination character and the hex value of the remainder is 0xF5.

Variable length characters

The decoded binary string's length in bits is given by :

$$\sum (LC) + rl$$

- where *C* is the number of *full* characters (excluding up to one partial character in the remainder)
- and *L* is the number of bits required to encode each of the full characters
- and *rl* is the number of bits implied by the value of the remainder encoded in the termination character.

For any Unicode variant and any source data, L is a multiple of 8 in the range 8 to 32.

Under UTF-8, L is most commonly 16. The average value of L depends on the distribution of source data values mapped to either Plane 0 (8 bits) or to one of the two higher planes (16 bits). For $Base_{15b}$ and below, all values are mapped to characters in the higher planes.

Under UTF-32, L is 32.

For a visual representation of this, see the Appendix19. Note that the ninth character in the sample $Base_{17b}$ encoding is 8 bits and in the LastResort font this glyph has a different appearance to the rest which represent characters of 16 bits.

Readability

Data encoded in the Asyntactic script is generally for use by computers and is not intended to be human-readable. Nonetheless, data encoded in the Asyntactic script may frequently appear in various human-readable formats. The script must therefore be capable of visual representation.

Font

Any font or fonts having sufficient quantity of glyphs may be specified to cover the full Asyntactic script code point range or just a subset of it. Since representations of the Asyntactic script are by definition independent of syntax, assigned characters need not have consistency between users, applications or systems.

Fall-back font

Many systems have a default mechanism[16] for viewing characters for which a font is not available. If a font has not been assigned to the specified code point range then the fall-back font will be used. The images in the Appendix19 are screen-captured from a desktop having Apple's LastResort[17] font installed.

Not for your eyes

In some circumstances it may be preferable to display an encoded text string in such a way that is clearly not intended for human consumption rather than, say, a Base64 string comprising familiar letters, digits and operators, which a human might unconsciously try to make sense of. The LastResort font certainly gives a visual impression that the presented text is data. See Appendix19.

16 http://en.wikipedia.org/wiki/Fallback_font
17 http://www.unicode.org/policies/lastresortfont_eula.html

Performance

UTF-8

The principal performance advantage that the Asyntactic script family has over Base 64 and Hex is its density of information per Character (efficiency). This chart14 shows information density17 under UTF-8 measured by Byte and by Character per Kibibit[18]. Lower numbers indicate greater efficiency.

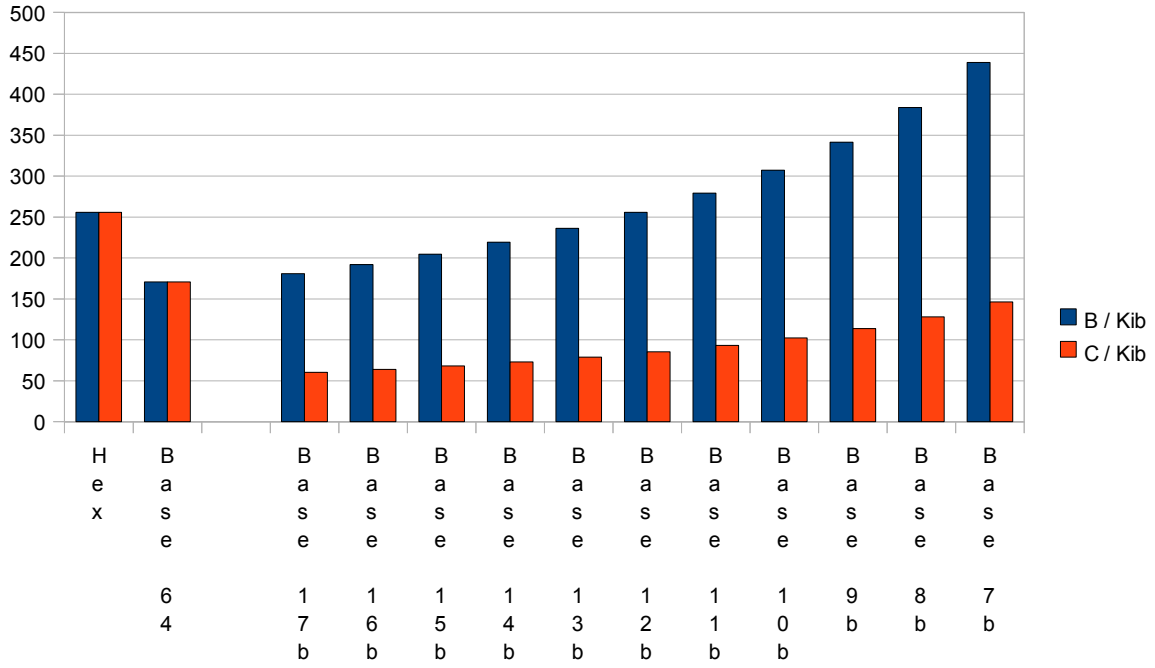

Table 5: Encoding information density by byte and character

In both performance classes, the most efficient member of the Asyntactic script encoding family is $Base_{17b.}9$

Information per byte

In comparison to Base 64, $Base_{17b}$ is expected to be slightly less efficient, putting around 6% more UTF-8 Bytes into the channel. A byte encoded in $Base_{16b}$ is around 12% less efficient than Base 64.

Information per character

In terms of information density per UTF-8 Character, $Base_{17b}$ is expected to be greater than 283% more efficient than Base 64. The equivalent score for $Base_{16b}$ is 266% improvement over Base 64.

Other Unicode variants

Less frequently used Unicode variants UTF-16 and UTF-32 expected results are given in the

18 http://en.wikipedia.org/wiki/Kibibit

Appendix17. They show broadly similar results and in most cases better Byte / Kib efficiency for the Asyntactic script family relative to Base64.

Expected / actual

The relative performance numbers given above are based on theory. Nevertheless, actual results19 of experimentation bear out the theory.

Caveats

These performance measurements are approximate. They measure the variable cost of each additional bit of information, but disregard the fixed cost of padding / terminating an encoded string. The longer the input string the closer these approximations are to actual.

An assumption was needed about the byte size of input data to UTF-16. The reported byte efficiency for UTF-16 result may not be reliable, but the character efficiency is not affected.

Side by side comparison

For a visual representation of the different encodings, see the Appendix19.

Signalling use of the Asyntactic script

Practical use of the Asyntactic script will be signalled through unambiguous contextual signposts within an application. These signposts are not part of the scope of this document. They will be published along with the specifications of the application which implements encoding in the Asyntactic script.

Applications

As a generic character encoding set, there can be many potential Unicode-capable applications for which the Asyntactic script would offer practical benefits.

Micro-blogging

Applications such as micro-blogging[19] typically constrain the number of characters, whilst leaving the number of bytes constrained only by the number of bytes required by Unicode. Under such conditions the $Base_{16b}$ family's character-wise efficiency should tend to bring it to the fore.

Non-standard syntax

There may also be applications where it is beneficial to display user-defined representations of data for which there is explicitly no syntactic representation.

19 http://en.wikipedia.org/wiki/Micro-blogging

Caution

Unicode support

Although Unicode is in increasingly widespread use, it is not ubiquitous. Some legacy systems which either cannot support Unicode or do support Unicode but have it disabled by default remain in operation. The Asyntactic script requires Unicode. It has neither a workaround nor graceful degradation for lack of Unicode support. It could break some non-Unicode applications.

This issue, will be mitigated over time, through replacement or upgrade of legacy systems. If the Asyntactic script ever becomes an unstoppable force in a particular domain, such legacy objects are likely to seem somewhat less unmovable.

Partial implementations of Unicode

It is feasible that some applications which claim to support Unicode may initially fail to support the Asyntactic script. For instance, the "Astral Planes[20]" are so infrequently used that some developers may have made the incorrect assumption that the BMP[21] plane represents the full extent of the Unicode range. This error can be mitigated by judicious deployment and careful testing.

Javascript is a case in point. Because the Unicode specification was finalised after Javascript's specification was finalised, some of Javascript's native functions do not support Unicode characters beyond 8 bits. In this case workarounds are available. For instance fixedFromCharCode[22] in the sample in the Appendix20 fixes the standard Javascript function FromCharCode, which is not multi-byte capable.

Missing font

Though characters in the Asyntactic script may look funny in the absence of a fall-back font, this state of affairs does not usually prevent the Asyntactic script from working. Although the previous statement appears to hold true for a few tested systems, it is not known to be universally true.

Conclusion

The Asyntactic script can be a useful addition to Base64 and other well known binary encoding formats. It might be particularly useful for particular applications such as micro-blogging, where efficiency of information per character is at a premium.

20 http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF

21 http://unicode.org/glossary/#basic_multilingual_plane

22https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/String/fromCharCode

Appendix

Performance tables

	UTF-8	B / Kib	C / Kib
	Hex	256	256
	Base 64	170.66	170.66
17	Base $_{17b}$	180.71	60.24
16	Base $_{16b}$	192	64
15	Base $_{15b}$	204.8	68.27
14	Base $_{14b}$	219.43	73.14
13	Base $_{13b}$	236.31	78.77
12	Base $_{12b}$	256	85.33
11	Base $_{11b}$	279.27	93.09
10	Base $_{10b}$	307.2	102.4
9	Base $_{9b}$	341.33	113.78
8	Base $_{8b}$	384	128
7	Base $_{7b}$	438.86	146.29

Table 6: comparison of encoding efficiencies under UTF-8

	UTF-16	B / Kib	C / Kib
	Hex	512	256
	Base 64	341.32	170.66
17	Base $_{17b}$	240.94	60.24
16	Base $_{16b}$	256	64
15	Base $_{15b}$	273.07	68.27
14	Base $_{14b}$	292.57	73.14
13	Base $_{13b}$	315.08	78.77
12	Base $_{12b}$	341.33	85.33
11	Base $_{11b}$	372.36	93.09
10	Base $_{10b}$	409.6	102.4
9	Base $_{9b}$	455.11	113.78
8	Base $_{8b}$	512	128
7	Base $_{7b}$	585.14	146.29

Table 7: comparison of encoding efficiencies under UTF-16

	UTF-32	B / Kib	C / Kib
	Hex	1024	256
	Base 64	682.64	170.66
17	Base $_{17b}$	240.94	60.24
16	Base $_{16b}$	256	64
15	Base $_{15b}$	273.07	68.27
14	Base $_{14b}$	292.57	73.14
13	Base $_{13b}$	315.08	78.77
12	Base $_{12b}$	341.33	85.33
11	Base $_{11b}$	372.36	93.09
10	Base $_{10b}$	409.6	102.4
9	Base $_{9b}$	455.11	113.78
8	Base $_{8b}$	512	128
7	Base $_{7b}$	585.14	146.29

Table 8: comparison of encoding efficiencies under UTF-32

Sequence of bases

Name	# of Characters	bits per Character
Binary Base 2	2	1
Base 4	4	2
Octal Base 8	8	3
Hex Base 16	16	4
Base 32	32	5
Base 64	64	6
$Base_{7b}$	128	7
$Base_{8b}$	256	8
$Base_{9b}$	512	9
$Base_{10b}$	1,024	10
$Base_{11b}$	2,048	11
$Base_{12b}$	4,096	12
$Base_{13b}$	8,192	13
$Base_{14b}$	16,384	14
$Base_{15b}$	32,768	15
$Base_{16b}$	65,536	16
$Base_{17b}$	131,072	17

Table 9: Sequence of bases: binary to Base17b

Experiment

A data source is encoded three different ways and the encoded characters placed side to side in a UTF-8 encoded web page, for visual comparison.

The data source is an image of "Larry" taken from the specification of Data URL[23]. The original source data was encoded in Base64. This was decoded to Hex and in turn encoded to $Base_{17b}$ using the Javascript in the Appendix20.

Results

$Base_{17b}$ has **65% fewer characters** than Base64.

Encoding	# of Characters	Compare to Base64
Hex	546	150.00%
(original) Base64	364	100.00%
$Base_{17b}$	129	35.00%

Visual comparison

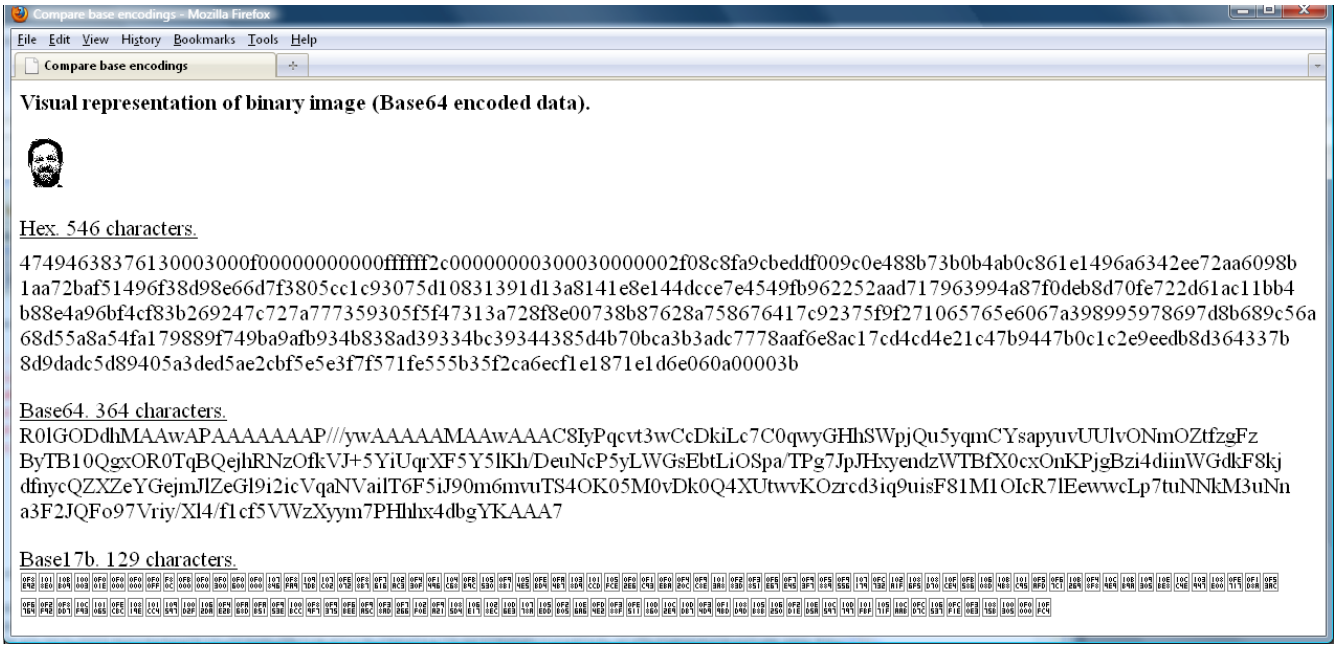

Illustration 1: side-by-side comparison of three encodings of the same data

The $Base_{1bb}$ data in the LastResort font looks neater on the page than does Base64 or Hex. Conveniently, this font allows on-screen inspection of each character's code point.

23 http://www.ietf.org/rfc/rfc2397.txt

Code sample

The Javascript code sample below encodes and decodes in $Base_{16b}$ and family. It is free to use under the MIT License[24].

```
/**
*  Base16b family encode / decode
*  http://base16b.org/lib/version/0.1/js/base16b.js
*  or http://base16b.org/lib/js/base16b.js
**/

/*
Copyright (c) 2009 Base16b.org

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
*/

var Base16b = {

        // private variables
        _asStart      : {value : 0x0000,        cp : 0xF0000},        // +UF0000 is the first code point in the Asyntactic script

        _noncont : function  () {
                var nc = [];    // array of cp : value mappings for the non-contiguous code points
                nc[0] = {value : 0xFFFE,    cp : 0xF80A};
                nc[1] = {value : 0xFFFF,    cp : 0xF80B};
                nc[2] = {value : 0x1FFFE,   cp : 0xF80C};
                nc[3] = {value : 0x1FFFF,   cp : 0xF80D};
                return nc;
        },

        // private methods
        _CharBytes : function (segmCP) { // return the number of bytes needed for the character. Usually 2.
                if (this._fixedCharCodeAt(segmCP,0) && this._fixedCharCodeAt(segmCP,1)) return 2; else return 1;
        },

        _invertVal : function  (segmVal, base) {
                // Two's complement of the value for this base
                return Math.pow(2, base) - (segmVal +1);
        },

        _fromCodePoint : function(segmCP, bytes) {
                // Map Code Point to a segment value as specified by the mapping table for this base in the Asyntactic script
                if (bytes === 2) return this._fixedCharCodeAt(segmCP,0) - this._asStart.cp;
                var i;
                for (i=0; i < this._noncont().length; i++) {                // handle non-contiguous code points for last two CPs in bases 16 and 17
                        if (this._fixedFromCharCode(this._noncont()[i].cp) === segmCP) {
                                return this._noncont()[i].value;
                        }
                }
        },

        _toCodePoint : function (segmVal, base) {
                // Map a segment value to the Code Point specified by the mapping table for this base in the Asyntactic script
                if  (base < 16) return this._asStart.cp + segmVal;
                var i;
                for (i=0; i < this._noncont().length; i++) {                // handle non-contiguous code points for bases 16 and 17
                        if (this._noncont()[i].value === segmVal) return this._noncont()[i].cp;
                }
                return this._asStart.cp + segmVal;
        },
```

24 http://www.opensource.org/licenses/mit-license.php

```
		_fixedFromCharCode : function (codePt) {
		// Fix the standard String.FromCharCode method to handle non-BMP unicode planes
		// https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/String/fromCharCode
	if (codePt > 0xFFFF) {
	    codePt -= 0x10000;
	    return String.fromCharCode(0xD800 + (codePt >> 10), 0xDC00 + (codePt & 0x3FF));
	}
	else {   return String.fromCharCode(codePt);  }
		},

		_fixedCharCodeAt : function (str, idx) {
		// https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/String/charCodeAt
	var code = str.charCodeAt(idx);
	 var hi, low;
	 if (0xD800 <= code && code <= 0xDBFF) { // High surrogate (could change last hex to 0xDB7F to treat high private surrogates as single characters)
	    hi = code;
	    low = str.charCodeAt(idx+1);
	    return ((hi - 0xD800) * 0x400) + (low - 0xDC00) + 0x10000;
	}
	 if (0xDC00 <= code && code <= 0xDFFF) { // Low surrogate
	    hi = str.charCodeAt(idx-1);
	   low = code;
	    return ((hi - 0xD800) * 0x400) + (low - 0xDC00) + 0x10000;
	}
	 return code;
		},

		// public method for encoding
		encode : function (inputArr, base)
/*
 Encode an array of pseudo-booleans (0 or 1)
 The specification of the encoding is documented elsewhere on this site. (Search Asyntactic script and Base16b.)
 */
{
		try {
			if (!(base >= 7 && base <= 17)) throw ('invalid encoding base: '+base);

			var resultArr		= [];
			var fullSegments	= Math.floor(inputArr.length / base);
			var remainBits				= inputArr.length - (fullSegments*base);
			var segment, bit;
			var segmstart;
			var segmVal ;				// construct the value of the bits in the current segment
			var currsegm;

			// convert the next segment of base numer of bits to decimal
			for (segment=0; segment < fullSegments; segment++) {		// input and output both read from left to right
				segmstart = base*segment;
				currsegm = inputArr.slice(segmstart, segmstart + base);
				// most significant bit at the start (left) / least significant bit at the end (right).
				for (bit = base-1, segmVal = 0; bit >= 0; bit--) {
					segmVal += (currsegm[bit] * Math.pow(2,(base-1) - bit));
				}
				resultArr[segment] = this._fixedFromCharCode(this._toCodePoint (segmVal, base));
			}
			// encode the termination character
			segmVal = 0;
			segmstart = base*segment;
			currsegm = inputArr.slice(segmstart);
			for (bit = remainBits-1; bit >= 0; bit--) {
				segmVal += (currsegm[bit] * Math.pow(2,(remainBits-1) - bit));
			}
			resultArr[segment]= this._fixedFromCharCode(this._toCodePoint(this._invertVal(segmVal, base), base));
			return resultArr.join('') ;
		}
		catch (e)
		{
			alert (e); return false;
		}
},

		// public method for decoding
		decode : function (inputStr)
		/*
		 Decode a string encoded in the Asyntactic script. Return an array of pseudo-booleans (0 or 1)
		 The specification of the encoding is documented elsewhere on this site. (Search Asyntactic script and Base16b.)
		 */
		{
			try {
				var resultArr = [];
				var termCharBytes = this._CharBytes(inputStr.slice(-2));
				var termCharCP = inputStr.slice(-termCharBytes);	// get the termination character
				var termCharVal		= this._fromCodePoint (termCharCP, termCharBytes);

				var bit = 17, base ;
				// decode the base from the termination character
				while (Math.floor(termCharVal / Math.pow(2, bit-1) ) === 0 && bit >=7 ) {bit--;}
				if (!(bit >= 7 && bit <= 17)) throw ('invalid encoding base');
				else base = bit;

				var segmVal ;
				var currCharBytes;
				var bytesUsed = 0;
```

```
                        var fullBytes = inputStr.length - termCharBytes;

                        while (bytesUsed < fullBytes) {             // decode the code point segments in sequence
                                currCharBytes = this._CharBytes(inputStr.slice(bytesUsed+2)); // taste before taking a byte
                                termCharCP = inputStr.slice(bytesUsed, bytesUsed + currCharBytes);
                                var segmVal = this._fromCodePoint (termCharCP, currCharBytes);
                                // most significant bit at the start (left) / least significant bit at the end (right).

                                for (bit = (currCharBytes*8)-1; bit >= 0; bit--) {
                                        resultArr.push( Math.floor( (segmVal / Math.pow(2,(bit)))%2));
                                }
                                bytesUsed += currCharBytes;
                        }



                        // remainder
                        var remainVal                           = this._invertVal(termCharVal, base);    // decode the remainder from the termination character

                        for (bit = (termCharBytes*8)-1; bit >= 0; bit--) {
                                resultArr.push( Math.floor( (remainVal / Math.pow(2,(bit)))%2));
                        }

                        return resultArr;

                }
                catch (e)
                {
                        alert (e); return false;
                }
        },

        // public method for counting Unicode characters
        trueLength : function (inputStr)
        /*
         Count the number of characters in a string.
         This function can handle stings of mixed BMP plane and higher Unicode planes.
         Fixes a problem with Javascript which incorrectly that assumes each character is only one byte.
         */
        {
                var strBytes  = inputStr.length;
                var strLength           = 0;
                var tallyBytes          = 0;
                try {
                        while (tallyBytes < strBytes) {
                                tallyBytes += this._CharBytes(inputStr.slice(tallyBytes, tallyBytes+2));
                                strLength +=1;
                        }
                        return strLength;
                }
                catch (e)
                {
                        alert (e); return false;
                }
        }

};
```