Base3z Encoding Model A Specification, version 1

Final Review Draft, released 15 December 2011

Notice: *This document is intended for public review and may be shared with other individuals or organizations on or after the release date only as a whole document with all text and legal notices intact, subject to the terms of the Limited Use License in Appendix B.*

Editor: Stephen Joyce, bitLab

Reviewers:
Please direct your comments regarding this document to: base3z@bitlab.com.

Abstract

The Base3z encoding method encodes binary data as sequences of Basic Multilingual Plane private-use code points, as defined in the Unicode® Standard. The encoded data can be represented as a sequence of UTF-8, UTF-16 or UTF-32 code units and subsequently decoded to yield the original binary data. This method requires minimal processing for both the encoding and decoding operations, and yields a 75% storage efficiency limit. Each datum encoding sequence includes type and encoding length information, enhancing parsing and search operation performance. The type system includes elements for creating complex structured data-text sequences, and supports application defined extensions.

Table of Contents

1 Introduction

The formats used for numeric data storage and communications vary widely in processing efficiency, program reusability and user accessibility. Using device memory images directly maximizes processing efficiency, but provides minimal levels of reusability and accessibility. Encoding numeric information as "readable" plain text provides moderate to high reusability and accessibility levels with relatively low processing efficiency. One alternative solution is to encode numeric and structure information using a type tagged "data-text" string format designed for efficient data transfer and storage.

Base3z encoding provides an extremely efficient method of platform independent data serialization using Unicode code points. Base3z encoding supports all common processor data types and bit sizes, very large integer and real numbers, text strings and indexed data arrays, type tags and complex data structuring, and inherent encoding error detection. The major applications of Base3z encoding include device or program configuration, logging and messaging, and HTML/XML content enhancement.

The programming languages used most often for real-time, embedded systems and platform software; native assembler, C and C++ do not provide standard data serialization support. Base3z encoding is a useful complement to these languages in particular, and offers a basis for high performance data-text processing, generally. This document defines a standard encoding model (A) designed to meet the requirements of these applications.

1.1 Base3z Encoding

The Basic Multilingual Plane defined in the Unicode® Standard contains a block of 6,400 code points, permanently reserved for the private use of any application, and referred to as the "private-use area." The Unicode® Standard does not define characters for these code points, allowing each to represent numerical data, custom characters or other information. The first section of this block is the set of 4,096 code points in the range U+E000 to U+EFFF, referred to here as the "E-block."

The most basic aspect of Base3z encoding is the mapping of any binary data set into a sequence of one or more E-block code points, each defined by the Boolean OR of the constant 0xE000 with a 12-bit data value from 0 to 0xFFF. The decode operation extracts the original data values defined by the Boolean AND of each code point with the constant 0x0FFF. The encoding of binary data as UTF-8, UTF-16 and UTF-32 code units, and the resulting storage efficiencies using this method is illustrated Figure 1.

None of the binary data sizes in common use today: 8-bits, 16-bits, 32-bits, 64-bits and 128-bits are a multiple of 12, which leaves unused encoding bits when single data values are processed. For example, encoding a 32-bit value requires 3 E-block code points, leaving 4 unused bits. Encoding a 64-bit value leaves 8 unused bits from 6 E-block code points. Data arrays and structures seldom measure a multiple of 12-bits in size, usually leaving unused bits in the last code point.

Figure 1: Base3z Data Encoding with E-Block Code Points

Binary data (12 bits/code point):
5AF, C84, 3B7, 210, 9DE, 060 …

Encoded data (leading zeros are significant):		Storage Efficiency:
UTF-8:	EE, 96, AF, EE, B2, 84, EE, 8E, B7, EE, 88, 90, EE, A7, 9E, EE, 81, A0 …	50%
UTF-16:	E5AF, EC84, E3B7, E210, E9DE, E060 …	75%
UTF-32:	0000E5AF, 0000EC84, 0000E3B7, 0000E210, 0000E9DE, 0000E060 …	37.5%

A key aspect of Base3z encoding is the use of these unused encoding bits in a systematic manner to specify data type and size information in each single value or data array encoding. The first code point in a single value encoding begins with a 4 or 8 bit type tag, followed by the initial data bits of the value. Any remaining data bits occupy additional code points as required for each type, leaving no unused encoding bits for single 8, 16, 32, 64 or 128 bit data values. The encoding of unsigned integer types is illustrated in Figure 2.

Figure 2: Base3z Encoding of Unsigned Integers

Size	Tag	Value	Code Point Sequence
8-bit:	0	12	E012
16-bit:	C0	1234	EC01, E234
32-bit:	2	12345678	E212, E345, E678
64-bit:	C4	123456789ABCDEF0	EC41, E234, E567, E89A, EBCD, EEF0
128-bit:	8	123456789ABCDEF0123456789ABCDEF0	E812, E345, E678, E9AB, ECDE, EF01, E234, E567, E89A, EBCD, EEF0

Data array and variable precision types use addition tag bits and code points as required to indicate both the type and size of the encoding. An important result of tagging all encoded data with type and size bits is the ability to quickly scan past individual encoded items within a code unit stream using the tag information to determine the encoding length of each item. Fixed precision types such as integer and floating point values have constant encoding lengths specific to each Unicode encoding form. Array and variable precision value encoding lengths can be derived using very simple formulas.

1.2 Design Features

The Base3z encoding method was originally developed at bitLab, LLC as an extension to the W3C Extensible Markup Language (XML) [**W3C XML 1.0**] for use in application configuration, messaging and logging operations. The original project goal was to create an efficient encoding for numeric attribute values and content of XML document elements. As the project evolved, the performance limitations of using XML as a structured data format led to a new goal: create a simple and efficient *structured data-text* technology for use in performance sensitive applications.

The key design features provided by Base3z encoding are as follows:

Internationalized

Base3z data encodings are compatible with the Unicode® Standard using any of the UTF-8, UTF-16 or UTF-32 encoding forms.

Wide application range

Base3z encoding supports all common binary data types, and provides an extensible system of type tags for building structured data-text applications.

High performance

Base3z encoding has a limiting spatial efficiency of 75% using the UTF-16 form. The encoding and decoding times are almost equal; achieving up to one-fourth the speed of a memcpy() operation using optimized C language code [see Section C.2].

Minimal complexity

Base3z encoding and decoding operations are simple to implement using a regular, compact and portable design that is fully compliant with this specification.

User accessible

Base3z encoded data is easy to identify within text streams as a sequence of E-block code points, and manual encoding or decoding is possible.

1.3 Definitions

The formal definitions of terms as they are used within this document are indicated by paragraphs indexed with [D] tags. These definitions apply throughout this document except where specifically noted otherwise.

[D1] A **codon** is a “well-formed” encoding of a Unicode code point as a sequence of UTF-8, UTF-16 or UTF-32 code units, as defined by the Unicode® Standard [**Unicode**]. Exactly one codon of each encoding form exists for every scalar value code point. Codons are not defined for UTF-16 surrogate code points. The codon identifier is the encoded code point value.

[D2] An **atom** is a sequence of one or more codons of the same encoding form that encode a Base3z data type as defined in this document. The atom type identifies the encoded data type.

[D3] A **codec** is a device or program that encodes Base3z data types as atoms, or decodes atoms to Base3z data types.

[D4] A **sequencer** is a device or program that uses a codec to encode or decode sequences of atoms according to the rules defined in this document. The codec may be part of the sequencer.

[D5] An **application** is a device or program that uses a codec or sequencer to encode or decode atoms. The codec or sequencer may be part of the application.

[D6] An **error** is the occurrence of a code unit, codon or atom sequence processing failure during the operation of a codec, sequencer or application.

1.4 Conformance

This document defines a number of conformance requirements for devices and programs that use Base3z encoding, indicated by paragraphs indexed with [C] tags.

In addition, when EMPHASIZED in this document, the key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL are to be interpreted as described in [**IETF RFC 2119**].

The primary conformance requirement is:

[C1] A conformant Base3z codec, sequencer or application MUST meet all requirements specified in this document that are relevant to the operation of the device or program.

2 Base3z Data Types

This section provides a formal specification of each Base3z data type and atom encoding using the EBNF style grammar defined in Appendix A. The rules defining code point classes, data type constants and atom codon patterns are indexed with [A] tags. The rules defining atom type groupings are indexed with [G] tags.

[C2] A codec MUST encode Base3z data types only as specified in this document.

[C3] A codec SHOULD be capable of decoding the length of all Base3z atoms that might occur as input to the device or program.

[C4] A codec MAY encode or fully decode a subset of the Base3z data types consistent with the requirements of a given device or program.

2.1 Code Point Classes

The Base3z code point set is identical to the code point set defined in the Unicode® Standard.

[A1] CodePoint := [U+0000, U+10FFFF]

All code points excluding the E-block and UTF-16 surrogate blocks are text code points.

[A2] TextCode := [U+0000, U+D7FF] | [U+F000, U+10FFFF]

The first text code point is often used to terminate string processing:

[A3] ZNulCode := U+0000

UTF-16 surrogate code points are members of either the leading or trailing surrogate block. The first code point value of each block is the class identifier constant for that block.

[A4] ZLeadingCode := U+D800

[A5] LeadingCode := [ZLeadingCode, U+DBFF]

[A6] ZTrailingCode := U+DC00

[A7] TrailingCode := [ZTrailingCode, U+DFFF]

[A8] ZSurrogateCode := ZLeadingCode

[A9] SurrogateCode := ZLeadingCode | ZTrailingCode

[C5] A codec SHOULD process LeadingCode and TrailingCode code points in a manner that is consistent with the Unicode® Standard.

The E-block code points are used for encoding binary data. The first code point value of this block is the class identifier for that block.

[A10] ZDataCode := U+E000

[A11] DataCode := [ZDataCode, U+EFFF]

The remaining Basic Multilingual Plane private-use area from U+F000 to U+F9FF is reserved for application specific use. The first code point value of this block is the class identifier for the block.

[A12] ZPrivateCode := U+F000

[A13] PrivateCode := [ZPrivateCode, U+F8FF]

A Unicode scalar value is either a “text” or “data” code point.

[A14] ScalarValue := TextCode | DataCode

All integer values greater than 10FFFF hexadecimal are invalid as code points. The first such code point value is the class identifier for these values.

[A15] ZInvalidCode := 110000H

2.2 Atom Starting Codons

The starting codon of an atom encodes a DataCode (E-block) code point with a 1 to 3 nibble type tag following the high order ‘E’ nibble. Zero or more additional codons encode the remainder of the atom. The starting code points of all data atoms are organized in Table 1. The left column indexes the first type tag nibble for the atom type or types in each row.

The “EC**” row contains the atom types with 2 or 3 nibble type tags. Nibbles marked with a “*” are further defined within the blocks of this row. The “ECA*” block includes a set of variable precision atom types. The “ECB*” block contains a set of fixed precision numeric array atom types.

All starting codon nibbles marked with “x” encode data, while nibbles marked with “s” encode application defined atom status or sub-type information. Substituting zeros for these nibbles produces the corresponding Base3z data type constants.

Table 1: Base3z Atom Starting Code Points

E0xx	Uns8														
E1xx	Int8														
E2xx	Uns32														
E3xx	Int32														
E4xx	Flt32														
E5xx	Dec32														
E6xx	Ptr32														
E7xx	Off32														
E8xx	Uns128														
E9xx	Int128														
EAxx	Flt128														
EBxx	Dec128														
EC**	Uns16 EC0x	Int16 EC1x	Seg16 EC2x	Off16 EC3x	Uns64 EC4x	Int64 EC5x	Flt64 EC6x	Dec64 EC7x	Ptr64 EC8x	Off64 EC9x	VP ECA*	Num[] ECB*	Data[] ECCs	Atom[] ECDs	Text[] ECEs	Char[] ECFs
EDxx	Symbol														
EExx	Enumerated														
EFxx	Customized														

2.3 Atom Encoding Lengths

Atom encoding lengths are measured in codons, and in code units. The abstract length is the number of codons specified by the atom encoding rules. The physical length in code units is a function of both the atom encoding rules and the selected encoding form. Codec operations are ultimately based upon atom physical lengths. The Base3z atom encoding lengths are summarized in Table 2.

Enumerated and fixed precision types have constant lengths, while variable precision and numeric array type lengths are simple functions of the data type and number of elements. These atoms are encoded using only DataCode (E-block) codons. The code unit length of these atoms is equal to the codon length using UTF-16 or UTF-32 encoding, and is 3 times that value using UTF-8 encoding.

TextArray and AtomBlock atoms contain non-DataCode codons, which vary in individual code unit lengths using UTF-8 and UTF-16 encoding. The size tags of these atoms specify the content length in code units to avoid scanning the content to determine the physical length.

CharArray atoms contain a character byte string appended to an array header that specifies the content length in bytes. The physical length of these atoms may include a trailing alignment byte for UTF-16 encodings.

TextString atoms are TextCode only codon sequences. Generally, the physical length of these atoms must be determined by content scanning.

Table 2: Base3z Atom Encoding Lengths

Enumerated	Enumerated, Customized	1 data codon
Fixed Precision Numeric, Position	**Int8, Uns8**	1 data codon
	Int16, Uns16 **Seg16, Off16**	2 data codons
	Dec32, Flt32, Int32, Uns32 **Ptr32, Off32**	3 data codons
	Dec64, Flt64, Int64, Uns64 **Ptr64, Off64**	6 data codons
	Dec128, Flt128, Int128, Uns128	11 data codons
Variable Precision P = precision multiple	**UnsVP, IntVP, FltVP, DecVP**	2 + (8P+2)/3 data codons
	BitString	2 + (bits+11)/12 data codons
	BCDString	4|7 + (digits+2)/3 data codons
Arrays N = array size S = member nibbles	**Fixed Precision Arrays**	4|7 + (SN+2)/3 data codons
	Variable Precision Arrays	5|8 + (8PN +2)/3 data codons
	DataBlock	4|7 + N data codons
	AtomBlock	4|7 data codons + N code units
	TextArray	4|7 data codons + N code units
	CharArray	4|7 + (2N+2)/3 data codons
	Symbol	1 data codon + N code units
TextCode[n]	**TextString**	Scanned code units

2.4 Atom Encoding and Byte Ordering

A Base3z encoding of any binary data value maps data value nibbles to code point nibbles in big-endian order. This mapping choice allows encoded integers of the same type to be sorted directly “as text” without decoding, alone or mixed within a plain text sequence.

Base3z encodings of numeric array types map the nibbles of each data element to code point nibbles in big-endian order, and encodes these elements in ascending array index order.

This logical data-nibble to codon-nibble mapping is independent of the physical byte ordering of a given UTF-16 or UTF-32 code unit stream.

2.4.1 Byte Order Marking

The correct decoding of a code unit stream to a sequence of Unicode scalar values requires knowledge of the Unicode encoding scheme used to produce the stream. The UTF-16 and UTF-32 *encoding forms* have multiple schemes based upon the physical byte ordering used, either big endian or little endian.

The Unicode® Standard provides guidelines for using the U+FEFF code point as a "byte order mark" (BOM) for determining the intended physical byte ordering of a code unit sequence. The byte order of this code point will be {0xFE, 0xFF} when big-endian schemes are used, and {0xFF, 0xFE} otherwise. When the U+FEFF code point is encoded and then decoded using opposite endian schemes, the decode operation produces the reserved code point U+FFFE, indicating the byte swap error.

[A16] ZNativeEndian := U+FEFF

[A17] ZByteSwapped := U+FFFE

2.4.2 Invalid Code Points

The Unicode® Standard provides guidelines for handling ill-formed code unit (sub) sequences by substituting one or more "replacement" characters for the erroneous data in the output sequence.

[A18] ZReplacedCode := U+FFFD

[C6] A codec or application MAY substitute ZReplacedCode characters for ill-formed code unit sub-sequences when decoding the content of TextArray or TextString atoms.

This specification defines additional techniques for handling Base3z processing errors in Section [3.4].

[C7] An application SHOULD mark or otherwise identify ill-formed codons and non-conformant atom encodings that are exchanged with other devices or programs.

2.5 Fixed Precision Numeric Atoms

Fixed precision numeric atoms encode 8-bit, 16-bit, 32-bit, 64-bit or 128-bit signed integers, unsigned integers, or floating-point numbers encoded as IEEE 754-2008 binary or decimal interchange formatted values [**IEEE STD 754**].

[C8] Applications SHOULD only encode IEEE 754-2008 interchange formatted floating-point values as Base3z floating-point atom types, if possible, to maximize data reuse.

[C9] Applications MUST indicate that non-IEEE 754-2008 interchange formatted floating-point values are encoded as Base3z floating-point atoms as specified in [Section 3.7.4].

2.5.1 8-bit Numbers

A single 8-bit numeric value is encoded as a single DataCode codon atom containing a single nibble type tag of 0 or 1, followed by 2 data nibbles “xx” in big-endian order.

8-bit Unsigned Integers:

[A19] ZUns8 := U+E000

[A20] Uns8 := U+E0xx

8-bit Signed Integers:

[A21] ZInt8 := U+E100

[A22] Int8 := U+E1xx

2.5.2 16-bit Numbers

A single 16-bit numeric value is encoded as a 2 DataCode codon atom containing a 2 nibble type tag of 0xC0 or 0xC1, followed by 4 data nibbles “x xxx” in big-endian order.

16-bit Unsigned Integers:

[A23] ZUns16 := U+EC00

[A24] Uns16 := U+EC0x U+Exxx

16-bit Signed Integers:

[A25] ZInt16 := U+EC10

[A26] Int16 := U+EC1x U+Exxx

2.5.3 32-bit Numbers

A single 32-bit numeric value is encoded as a 3 DataCode codon atom containing a single nibble type tag of 2, 3, 4 or 5, followed by 8 data nibbles “xx xxx …” in big-endian order.

32-bit Unsigned Integers:

[A27] ZUns32 := U+E200

[A28] Uns32 := U+E2xx U+Exxx{2}

32-bit Signed Integers:

[A29] ZInt32 := U+E300

[A30] Int32 := U+E3xx U+Exxx{2}

32-bit Binary Floating Point Numbers:

[A31] ZFlt32 := U+E400

[A32] Flt32 := U+E4xx U+Exxx{2}

32-bit Decimal Floating Point Numbers:

[A33] ZDec32 := U+E500

[A34] Dec32 := U+E5xx U+Exxx{2}

2.5.4 64-bit Numbers

A single 64-bit numeric value is encoded as a 6 DataCode codon atom containing a 2 nibble type tag of 0xC4, 0xC5, 0xC6 or 0xC7, followed by 16 data nibbles “x xxx …” in big-endian order.

64-bit Unsigned Integers:

[A35] ZUns64 := U+EC40

[A36] Uns64 := U+EC4x U+Exxx{5}

64-bit Signed Integers:

[A37] ZInt64 := U+EC50

[A38] Int64 := U+EC5x U+Exxx{5}

64-bit Binary Floating Point Numbers:

[A39] ZFlt64 := U+EC60

[A40] Flt64 := U+EC6x U+Exxx{5}

64-bit Decimal Floating Point Numbers:

[A41] ZDec64 := U+EC70

[A42] Dec64 := U+EC7x U+Exxx{5}

2.5.5 128-bit Numbers

A single 128-bit numeric value is encoded as an 11 DataCode codon atom containing a single nibble type tag of 8, 9, A or B, followed by 32 data nibbles “xx xxx …” in big-endian order.

128-bit Unsigned Integers:

[A43] ZUns128 := U+E800

[A44] Uns128 := U+E8xx U+Exxx{10}

128-bit Signed Integers:

[A45] ZInt128 := U+E900

[A46] Int128 := U+E9xx U+Exxx{10}

128-bit Binary Floating Point Numbers:

[A47] ZFlt128 := U+EA00

[A48] Flt128 := U+EAxx U+Exxx{10}

128-bit Decimal Floating Point Numbers:

[A49] ZDec128 := U+EB00

[A50] Dec128 := U+EBxx U+Exxx{10}

2.5.6 Fixed Precision Numeric Atom Groups

The fixed precision numeric atom groups are:

[G1] Unsigned := Uns8 | Uns16 | Uns32 | Uns64 | Uns128

[G2] Integer := Int8 | Int16 | Int32 | Int64 | Int128

[G3] Float := Flt32 | Flt64 | Flt128

[G4] Decimal := Dec32 | Dec64 | Dec128

[G5] NumericAtom := Unsigned | Integer | Float | Decimal

2.6 Fixed Precision Numeric Array Atoms

A fixed precision numeric array atom encodes a vector of 8-bit, 16-bit, 32-bit, 64-bit or 128-bit integer or IEEE 754-2008 interchange formatted floating-point values. These atoms consist of a starting codon containing a 3 nibble type tag from 0xCAA to 0xCBF, followed by an Uns32 or Uns64 encoding of the array size N, followed by the encoded data nibbles “xxx …” of each array element in big-endian, ascending array index order. The encoding of an unsigned byte array is illustrated in Figure 3.

For any element size of S nibbles, the atom encodes 3 nibbles in every DataCode codon. The codon length L of a fixed precision numeric array atom of N elements is:

L = 1 + (3|6) + (SN+2)/3 …using integer division.

The minimum possible number of codons is used to encode the array elements, which are leading-justified in the codon sequence. Any unused nibbles in the last codon are set to zero.

Figure 3: Base3z Encoding of an Unsigned Byte Array

8-byte data array:	12, 34, 56, 78, 9A, BC, DE, F0, 12, 34
Atom type-size tag:	CAA (8-bit unsigned integer array)
Array size tag:	E200, E000, E00A (Uns32 = 10)
Code point sequence:	ECAA, E200, E000, E00A, E123, E456, E789, EABC, EDEF, E012, E34z (z = zero padding)

The data type constant and codon pattern rules for these atom types are:

2.6.1 8-bit Numeric Arrays

[A51] ZUns8Array := U+ECAA

[A52] Uns8Array := ZUns8Array (Uns32|Uns64) U+Exxx{(2N+2)/3}

[A53] ZInt8Array := U+ECAB

[A54] Int8Array := ZInt8Array (Uns32|Uns64) U+Exxx{(2N+2)/3}

2.6.2 16-bit Numeric Arrays

[A55] ZUns16Array := U+ECAC

[A56] Uns16Array := ZUns16Array (Uns32|Uns64) U+Exxx{(4N+2)/3}

[A57] ZInt16Array := U+ECAD

[A58] Int16Array := ZInt16Array (Uns32|Uns64) U+Exxx{(4N+2)/3}

2.6.3 32-bit Numeric Arrays

[A59] ZUns32Array := U+ECB0

[A60] Uns32Array := ZUns32Array (Uns32|Uns64) U+Exxx{(8N+2)/3}

[A61] ZInt32Array := U+ECB1

[A62] Int32Array := ZInt32Array (Uns32|Uns64) U+Exxx{(8N+2)/3}

[A63] ZFlt32Array := U+ECB2

[A64] Flt32Array := ZFlt32Array (Uns32|Uns64) U+Exxx{(8N+2)/3}

[A65] ZDec32Array := U+ECB3

[A66] Dec32Array := ZDec32Array (Uns32|Uns64) U+Exxx{(8N+2)/3}

2.6.4 64-bit Numeric Arrays

[A67] ZUns64Array := U+ECB6

[A68] Uns64Array := ZUns64Array (Uns32|Uns64) U+Exxx{(16N+2)/3}

[A69] ZInt64Array := U+ECB7

[A70] Int64Array := ZInt64Array (Uns32|Uns64) U+Exxx{(16N+2)/3}

[A71] ZFlt64Array := U+ECB8

[A72] Flt64Array := ZFlt64Array (Uns32|Uns64) U+Exxx{(16N+2)/3}

[A73] ZDec64Array := U+ECB9

[A74] Dec64Array := ZDec64Array (Uns32|Uns64) U+Exxx{(16N+2)/3}

2.6.5 128-bit Numeric Arrays

[A75] ZUns128Array := U+ECBC

[A76] Uns128Array := ZUns128Array (Uns32|Uns64) U+Exxx{(32N+2)/3}

[A77] ZInt128Array := U+ECBD

[A78] Int128Array := ZInt128Array (Uns32|Uns64) U+Exxx{(32N+2)/3}

[A79] ZFlt128Array := U+ECBE

[A80] Flt128Array := ZFlt128Array (Uns32|Uns64) U+Exxx{(32N+2)/3}

[A81] ZDec128Array := U+ECBF

[A82] Dec128Array := ZDec128Array (Uns32|Uns64) U+Exxx{(32N+2)/3}

2.6.6 Fixed Precision Numeric Array Atom Groups

The fixed precision numeric array atom groups are:

[G6] UnsignedArray := Uns8Array | Uns16Array | Uns32Array | Uns64Array | Uns128Array

[G7] IntegerArray := Int8Array | Int16Array | Int32Array | Int64Array | Int128Array

[G8] FloatArray := Flt32Array | Flt64Array | Flt128Array

[G9] DecimalArray := Dec32Array | Dec64Array | Dec128Array

[G10] NumericArray := UnsignedArray | IntegerArray | FloatArray | DecimalArray

2.7 Segment, Pointer and Offset Atoms

A segment atom encodes a 16-bit unsigned integer storage segment number. A pointer atom encodes a 32-bit or 64-bit unsigned integer storage location. An offset atom encodes a 16-bit, 32-bit or 64-bit signed integer storage location offset. The storage units and meaning of these atom values is entirely application dependent. No inherent relationship exists between a segment, pointer or offset value and other Base3z atoms. Decoded pointer and offset values can be relocated or "swizzled" according to a programming language data model or an application schema.

2.7.1 Segments

A single 16-bit segment value is encoded as a 2 DataCode codon atom containing a 2 nibble type tag of 0xC2, followed by 4 data nibbles "x xxx" in big-endian order.

[A83] ZSeg16 := U+EC20

[A84] Seg16 := U+EC2x U+Exxx

2.7.2 Pointers

A single 32-bit pointer value is encoded as a 3 DataCode codon atom containing a single nibble type tag of 0x6, followed by 8 data nibbles "xx xxx …" in big-endian order.

[A85] ZPtr32 := U+E600

[A86] Ptr32 := U+E6xx U+Exxx{2}

A single 64-bit pointer value is encoded as a 6 DataCode codon atom containing a 2 nibble type tag of 0xC8, followed by 16 data nibbles "x xxx …" in big-endian order.

[A87] ZPtr64 := U+EC80

[A88] Ptr64 := U+EC8x U+Exxx{5}

2.7.3 Offsets

A single 16-bit offset value is encoded as a 2 DataCode codon atom containing a 2 nibble type tag of 0xC3, followed by 4 data nibbles "x xxx" in big-endian order.

[A89] ZOff16 := U+EC30

[A90] Off16 := U+EC3x U+Exxx

A single 32-bit offset value is encoded as a 3 DataCode codon atom containing a single nibble type tag of 0x7, followed by 8 data nibbles “xx xxx …” in big-endian order.

[A91] ZOff32 := U+E700

[A92] Off32 := U+E7xx U+Exxx{2}

A single 64-bit offset value is encoded as a 6 DataCode codon atom containing a 2 nibble type tag of 0xC9, followed by 16 data nibbles “x xxx …” in big-endian order.

[A93] ZOff64 := U+EC90

[A94] Off64 := U+EC9x U+Exxx{5}

2.7.4 Position Atom Groups

The position atom groups are:

[G11] Pointer := Ptr32 | Ptr64

[G12] Offset := Off16 | Off32 | Off64

[G13] PositionAtom := Offset | Pointer | Seg16

2.8 Segment, Pointer and Offset Array Atoms

A segment array atom encodes a vector of 16-bit storage segment numbers. A pointer array atom encodes a vector of 32-bit or 64-bit storage locations. An offset array atom encodes a vector of 16-bit, 32-bit or 64-bit storage location offsets. These atoms consist of a starting codon containing a 3 nibble type tag from 0xCAA to 0xCBF, followed by an Uns32 or Uns64 atom encoding of the array size N, followed by the encoded data nibbles “xxx …” of each array element in big-endian, ascending array index order. The encoding length and data packing of these atoms are defined the same as fixed precision numeric atoms in Section 2.6.

The data type constant and codon pattern rules for these atoms are as follows:

2.8.1 Segment Arrays

[A95] ZSeg16Array := U+ECAE

[A96] Seg16Array := ZSeg16Array (Uns32|Uns64) U+Exxx{(4N+2)/3}

2.8.2 Pointer Arrays

[A97] ZPtr32Array := U+ECB4

[A98] Ptr32Array := ZPtr32Array (Uns32|Uns64) U+Exxx{(8N+2)/3}

[A99] ZPtr64Array := U+ECBA

[A100] Ptr64Array := ZPtr64Array (Uns32|Uns64) U+Exxx{(16N+2)/3}

2.8.3 Offset Arrays

[A101] ZOff16Array := U+ECAF

[A102] Off16Array := ZOff16Array (Uns32|Uns64) U+Exxx{(4N+2)/3}

[A103] ZOff32Array := U+ECB5

[A104] Off32Array := ZOff32Array (Uns32|Uns64) U+Exxx{(8N+2)/3}

[A105] ZOff64Array := U+ECBB

[A106] Off64Array := ZOff64Array (Uns32|Uns64) U+Exxx{(16N+2)/3}

2.8.4 Position Array Atom Groups

The position array atom groups are:

[G14] PointerArray:= Ptr32Array | Ptr64Array

[G15] OffsetArray := Off16Array | Off32Array | Off64Array

[G16] PositionArray := OffsetArray | PointerArray | Segment16Array

2.9 Variable Precision Numeric Atoms

A variable precision numeric atom encodes an integer or IEEE 754-2008 formatted value of bit-size 32P, from 32 (0x20) to 131,072 (0x20000) bits. These atoms consist of a starting codon containing a 3 nibble type tag 0xCA1, 0xCA2, 0xCA3 or 0xCA4, followed by a second codon containing a 3 nibble precision multiple P, with zero representing 0x1000, followed by the encoded data nibbles “xxx-xxx-xx …” in big-endian order.

The codon length of a variable precision atom of bit-size 32P is:

L = 2 + (8P+2)/3 …using integer division.

The data type constant and codon pattern rules for these atoms are as follows:

2.9.1 Variable Precision Unsigned Integers

[A107] ZUnsVP := U+ECA1

[A108] UnsVP := ZUnsVP U+Eppp U+Exxx{(8P+2)/3}

2.9.2 Variable Precision Signed Integers

[A109] ZIntVP := U+ECA2

[A110] IntVP := ZIntVP U+Eppp U+Exxx{(8P+2)/3}

2.9.3 Variable Precision Binary Floating Point Numbers

[A111] ZFltVP := U+ECA3

[A112] FltVP := ZFltVP U+Eppp U+Exxx{(8P+2)/3}

2.9.4 Variable Precision Decimal Floating Point Numbers

[A113] ZDecVP := U+ECA4

[A114] DecVP := ZDecVP U+Eppp U+Exxx{(8P+2)/3}

2.9.5 Variable Precision Numeric Atom Group

The variable precision atom group is:

[G17] VPNumericAtom := UnsVP | IntVP | FltVP | DecVP

2.10 Variable Precision Numeric Array Atoms

A variable precision numeric array atom encodes a vector of variable precision values of bit-size 32P, from 32 (0x20) to 131,072 (0x20000) bits. The atom consists of a starting codon containing a 3 nibble type tag 0xCA5, 0xCA6, 0xCA7 or 0xCA8, followed by a 3 nibble encoding of P, with zero representing 0x1000, followed by an Uns32 or Uns64 atom encoding of the array size N, followed by the encoded data nibbles “xxx-xxx-xx …” in big-endian, ascending array index order.

The codon length of a variable precision array atom of member bit-size 32P and array size N is:

L = 2 + (3|6) + (8PN+2)/3 data codons

The encoding of a 32-bit standard sequence precision element array is illustrated in Figure 4.

The data type constant and codon pattern rules for these atoms are as follows:

2.10.1 Variable Precision Unsigned Integer Arrays

[A115] ZUnsVPArray := U+ECA5

[A116] UnsVPArray := ZUnsVPArray U+Eppp (Uns32|Uns64) U+Exxx{(8PN+2)/3}

2.10.2 Variable Precision Signed Integer Arrays

[A117] ZIntVPArray := U+ECA6

[A118] IntVPArray := ZIntVPArray U+Eppp (Uns32|Uns64) U+Exxx{(8PN+2)/3}

2.10.3 Variable Precision Binary Floating Point Arrays

[A119] ZFltVPArray := U+ECA7

[A120] FltVPArray := ZFltVPArray U+Eppp (Uns32|Uns64) U+Exxx{(8PN+2)/3}

2.10.4 Variable Precision Decimal Floating Point Arrays

[A121] ZDecVPArray := U+ECA8

[A122] DecVPArray := ZDecVPArray U+Eppp (Uns32|Uns64) U+Exxx{(8PN+2)/3}

Figure 4: Base3z Encoding of a 32-bit Variable Precision Binary Floating Point Array

Encoded values:	0.123456, 789.012, -345.678, -901.234, 567.890
Atom type tag:	CA7 (variable precision binary FP array)
Precision tag:	001 (32-bits)
Array length tag:	E200, E000, E005 (Uns32 = 5)
Code point sequence:	ECA7, E001, E200, E000, E005, E80D, E6FC, E3DC, E540, E454, E4C9, ED6A, ECC3, EFA4, EE61, EC4F, E6F8, E0D4, E4zz (z = zero padding)

2.10.5 Variable Precision Numeric Array Atom Group

The variable precision array atom group is:

[G18] VPNumericArray := UnsVPArray | IntVPArray | FltVPArray | DecVPArray

2.11 Bit Strings and BCD Strings

2.11.1 Bit Strings

A BitString atom encodes a bit string of size S, from 1 to 0x1000. The atom consists of a starting DataCode codon containing the 3 nibble type tag 0xCA0, followed by a 3 nibble encoding of S, with zero representing 0x1000, followed by a sequence of (S+11)/12 DataCode codons containing the bits “bbbbbbbbbbb, …” in ascending array index order. The minimum possible number of codons is used to encode the array data, which is leading-justified in the codon sequence. Unused bits in the last data codon are set to zero.

The data type constant and codon pattern rules for these atoms are as follows:

[A123] ZBitString := U+ECA0

[A124] BitString := ZBitString U+Esss U+1110bbbbbbbbbbbb{(S+11)/12}

2.11.2 BCD Strings

A BCDString atom encodes a packed binary coded decimal string. The atom consists of a starting codon containing the 3 nibble type tag 0xCA9, followed by an Uns32 or Uns64 atom encoding of the string length N, followed by the encoded digit values "ddd …" from the symbol set defined in Table 3.

The data encoding consists of an optional leading negation symbol, a real number with a whole or a fractional part indicated by a leading decimal point symbol or both, and an optional signed base 10 exponent. The real number and exponent are encoded in big-endian order.

Table 3: Decimal Number and Symbol Set

Nibble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Digits	0	1	2	3	4	5	6	7	8	9						
Symbol											Decimal point	.				
												Base 10 exponent	e			
													Negative	-		
														Ratio	/	
															Trailing blank	_
																reserved *

The minimum possible number of codons is used to encode the digits and symbols, which are leading-justified in the codon sequence and padded with trailing "blank" nibbles as needed in the last codon. The encoding of these atoms is illustrated in Figure 5.

The data type constant and codon pattern rules for these atoms are as follows:

[A125] ZBCDString := U+ECA9

[A126] BCDString := ZBCDString (Uns32|Uns64) U+Eddd{(N+2)/3}

The encoding of a decimal digit string does not contain locale or formatting information.

[C10] Applications SHOULD NOT use the reserved symbol value 0xF, which is reserved for definition in a future version of this specification.

Figure 5: Base3z Encoding of Decimal and Scientific Numbers

Decimal number:	-12,345,678.90
BCD type tag:	CA9
Digit count:	E200, E000, E00C (Uns32 = 12)
Code point sequence:	ECA9, E200, E000, E00C, ED12, E345, E678, EF90
Scientific number:	$3.4567890 \times 10^{-12}$
BCD type tag:	CA9
Digit count:	E200, E000, E00D (Uns32 = 13)
Code point sequence:	ECA9, E200, E000, E00D, E3F4, E567, E890, EED1, E2BB

2.12 Data Block Atoms

A DataBlock atom contains a vector of DataCode codons. The atom consists of a starting codon containing the 2 nibble type tag 0xCC and a nibble "s" reserved for application defined status bits, followed by an Uns32 or Uns64 atom encoding of the block size N *codons*, followed by the N DataCode codons. The encoding of a DataBlock atom is illustrated in Figure 6.

The data type constant and codon pattern rules for these atoms are as follows:

[A127] ZDataBlock := U+ECC0

[A128] DataBlock := U+ECCs (Uns32|Uns64) DataCode{N}

Figure 6: Base3z Encoding of a DataBlock Atom

Block contents:	E001, E002, E003, E004
Type tag:	CC (DataBlock)
Status flags tag:	0 (application defined)
Codon count tag:	E200, E000, E004 (Uns32 = 4)
Code point sequence:	ECC0, E200, E000, E004, E001, E002, E003, E004

The encoding and decoding process for DataBlock atom content is application defined. This content can be entirely opaque to uniformed applications. The array header status nibble, related atoms or application state can indicate the expected processing of the atom.

2.13 Atom Block Atoms

AtomBlock atoms contain other Base3z atoms as a vector of code units. These atoms consist of a starting codon containing the 2 nibble type tag 0xCD and a nibble "s" used for application defined status bits, followed by an Uns32 or Uns64 atom encoding of the block size N *code units*, followed by the N code units. An example of an AtomBlock atom encoding is illustrated in Figure 7.

The data type constant and codon pattern rules for these atoms are as follows:

[A129] ZAtomBlock := U+ECD0

[A130] AtomBlock := U+ECDs (Uns32|Uns64) Ux{N}

Figure 7: Base3z Encoding of an AtomBlock Atom

Block contents:	EC1F, EFFF, EC10, E000, EC10, E001 (Int16=-1, Int16=0, Int16=1)
Atom type tag:	CD (AtomBlock)
Status flags tag:	0 (application defined)
Array length tag:	E200, E000, E006 (Uns32= 6)
Code point sequence:	ECD0, E200, E000, E006, EC1F, EFFF, EC10, E000, EC10, E001

AtomBlock atoms can contain a sequence of any Base3z atom types, including other AtomBlock atoms. The expected processing of the atom can be indicated using the array header status nibble, related atoms or application state.

2.13.1 Well Formed Atom Blocks

A *well-formed* AtomBlock atom contains a sequence of zero or more Base3z atoms with a total length exactly equal to the size of the AtomBlock. ZNulCode strings or other atoms can be used to adjust the AtomBlock content when other contained atom sizes change.

[C11] AtomBlock atoms SHOULD be well formed and contain only valid Base3z atom encodings following a related series of codec write operations on the atom content.

2.14 Text Atoms

Text atoms are arrays of Unicode scalar values, arrays of legacy byte and multi-byte characters, application defined symbols, or strings of un-encapsulated TextCode codons.

2.14.1 Text Arrays

A TextArray atom contains a vector of code units that encode a sequence of ScalarValue code points. The atom consists of a starting codon containing a 2 nibble type tag 0xCE and a nibble "s" reserved for

application defined status bits, followed by an Uns32 or Uns64 atom encoding of the array size N *code units*, followed by the ScalarValue codon sequence of length N code units. The encoding of a TextArray atom is illustrated in Figure 8.

The data type constant and codon pattern rules for these atoms are as follows:

[A131] ZTextArray := U+ECE0

[A132] TextArray := U+ECEs (Uns32|Uns64) Ux{N}

Figure 8: Base3z Encoding of a TextArray Atom

Scalar values:	42, 61, 73, 65, 33, 7A	"Base3z"
Type tag:	CE (TextArray)	
Status flags tag:	0 (application defined)	
Code unit count tag:	E200, E000, E006 (Uns32 = 6)	
Code point sequence:	ECE0, E200, E000, E006, 42, 61, 73, 65, 33, 7A	
UTF-8 encoding:	EE, B3, A0, EE, 88, 80, EE, 80, 80, EE, 80, 86, 42, 61, 73, 65, 33, 7A	
UTF-16 encoding:	ECE0, E200, E000, E006, 0042, 0061, 0073, 0065, 0033, 007A	

DataCode codons within the content of application tagged TextArray atoms are *always* defined as private-use E-block code points, allowing legacy E-block usage to be preserved. Applications can also process these codons as Base3z atoms as a private use case. Related atoms or other application state can further define the expected processing of the atom as indicated by the status value.

[C12] The use of Base3z atoms as TextArray content is RECOMMENDED only for linear data-text constructs that are processed primarily as strings. Use AtomBlock atoms to construct more complex data-text structures.

[C13] A TextArray SHOULD contain only valid ScalarValue encodings following a related series of codec write operations on the array.

2.14.2 Character Arrays

A CharArray atom encodes a sequence of byte or multiple byte characters as a vector of byte values. The atom consists of a starting codon containing a 2 nibble type tag 0xCF and a nibble "s" reserved for application defined status bits, followed by an optional Uns16 atom code page identifier, followed by an Uns32 or Uns64 atom encoding of the array size N *bytes*, followed by the encoded bytes. The encoding of a CharArray atom is illustrated in Figure 9.

The data type constant and codon pattern rules for these atoms are as follows:

[A133] ZCharArray := U+ECF0

[A134] CharArray := U+ECF0 (Uns16) (Uns32|Uns64) Exxx{(2N+2)/3}

Figure 9: Base3z Encoding of a CharArray Atom

Character bytes:	42, 61, 73, 65, 33, 7A	"Base3z"
Type tag:	CF	(CharArray)
Status flags tag:	0	(application defined)
Code page tag:	EC00, E4E4	(Windows 1252 "Latin-1")
Character count tag:	E200, E000, E006	(Uns32 = 6)
Code point sequence:	ECF0, EC00, E4E4, E200, E000, E006, E426, E173, E653, E37A	

The character set corresponding to a code page value is application defined. The absence of a code page identifier indicates a default code page defined by the application.

Legacy code-page character byte strings can be encoded with a storage efficiency of 75% using UTF-16 CharArray atoms.

2.14.3 Symbols

A Symbol atom encodes a short sequence of ScalarValue code points. The atom consists of a starting codon containing a single nibble type tag 0xD and a 2-nibble sequence length N, from 0 to 255 *code units*, followed by the N code units. The encoding of a Symbol atom is illustrated in Figure 10.

The data type constant and codon pattern rules for these atoms are as follows:

[A135] ZSymbol := U+ED00

[A136] Symbol := U+EDnn Ux{N}

Figure 10: Base3z Encoding of a Symbol Atom

Scalar values:	42, 61, 73, 65, 33, 7A	"Base3z"
Type tag:	D (Symbol)	
Code unit count:	6	
Code point sequence:	ED06, 42, 61, 73, 65, 33, 7A	
UTF-8 encoding:	EE, B4, 86, 42, 61, 73, 65, 33, 7A	
UTF-16 encoding:	ED06, 0042, 0061, 0073, 0065, 0033, 007A	

2.14.4 Text Strings

A TextString atom consists of a sequence of TextCode only codons. When outside of the content of a TextArray or AtomBlock atom, a TextString is bounded on either end by a DataCode codon or the code unit stream limits. Within a container atom, a TextString is also limited to the atom content bounds. Generally, the length of these atoms must be determined by content scanning.

The codon pattern rule for these atoms is:

[A137] TextString := TextCode{N}

2.14.5 Legacy Private-Use E-block Codons

When an application must process Base3z atoms and "legacy" text including E-block codons in the same codon stream, the basic (re) design options are:

Mark the legacy E-block codons or encapsulate them in TextArray or CharArray atoms.

Remap the legacy E-block codons to the PrivateCode block or other non-DataCode codons.

The first option is usually sufficient to preserve the original text content and processing intact.

2.14.6 Text Atom Group

The text atom group is:

[G19] TextAtom := TextArray|CharArray | Symbol | TextString

2.15 Enumerated Atoms

Enumerated atoms are single DataCode codons that represent constant values, or combine with other atoms to "compound" their meaning as *structured atom sequences*. An enumerated atom is identified by its code point value. The set of enumerated atoms defined for this version of the specification are defined in Section 3.

The data type constant and codon pattern rules for these atoms are as follows:

[A138] ZEnumerated := U+EE00

[A139] Enumerated := [ZEnumerated, U+EEFF]

[C14] Applications MUST use Enumerated atoms only in a manner that is consistent with the definitions and rules contained in this document.

2.15.1 Custom Enumerated Atoms

Customized atoms are enumerated atoms in a code point range reserved for definition by applications or protocols, subject to the rules defined in this document.

The data type constant and codon pattern rule for these atoms are as follows:

[A140] ZCustomized := U+EF00

[A141] Customized := [ZCustomized, U+EFFF]

[C15] Applications MUST use Customized atoms only in a manner that is consistent with the definitions and rules contained in this document.

2.16 Atom Encoding Length Limits

The atom encoding length limits supported by a codec or application for Base3z will depend upon the design requirements, available memory, the atom type, and the encoding scheme. The largest scalar atom type, the VariablePrecision atom, has a maximum encoding length of (only) 10,922 data codons.

Array atom sizes can be encoded using Uns32 or Uns64 atoms, which allows atom encoding lengths larger than existing storage system ranges to be specified. Encoding “models” can be used to define array limits for specific device or program storage system constraints.

[C16] A codec MUST decode an Uns32 atom size for a given array type if it decodes an Uns64 atom size for the same array type.

[C17] A codec or application MUST indicate an error condition when an array atom size operating limit is exceeded during encode or decode operations.

[C18] A codec or application SHOULD publish array atom size operating limits.

2.16.1 Medium Memory Model

The medium memory model supports atoms up to 2^{32} or 4 gigabytes in length. Allowing for 7 codon array headers (type+Uns64 tags), the resulting size limits for UTF-8, UTF-16 and UTF-32 encoded arrays

are presented in Table 4. With a single exception (*), all of these size limits are less than 0x100000000, which allows the use of 32-bit size parameters and operations by codecs and applications.

Table 4: Medium Model Array Size Limits

Array Type	UTF-8 Encoding	UTF-16 Encoding	UTF-32 Encoding
Z…8Array	0x7FFFFFF5	0xBFFFFFF5	0x5FFFFFF5
Z…16Array	0x3FFFFFFA	0x5FFFFFFA	0x2FFFFFFA
Z…32Array	0x1FFFFFFD	0x2FFFFFFD	0x17FFFFFD
Z…64Array	0x0FFFFFFE	0x17FFFFFE	0x0BFFFFFE
Z…128Array	0x07FFFFFF	0x0BFFFFFF	0x05FFFFFF
ZBCDString	0xFFFFFFEA	0xFFFFFFFF*	0xBFFFFFEB
ZDataBlock	0x5555554E	0x7FFFFFF9	0x3FFFFFF9
ZAtomBlock	0xFFFFFFEB	0x7FFFFFF9	0x3FFFFFF9
ZTextArray	0xFFFFFFEB	0x7FFFFFF9	0x3FFFFFF9
ZCharArray	0x7FFFFFF5	0xBFFFFFF5	0x5FFFFFF5

* The actual limit is 0x17FFFFEB.

The UTF-32 encoding form imposes the smallest limits for all array types, setting the upper bounds for array sizes that can be converted among all 3 encoding forms.

2.16.2 Large Memory Model

The large memory model supports atoms up to 2^{64} or 16 exabytes in length. The corresponding set of array size limits are presented in Table 5.

Table 5: Large Model Array Size Limits

Array Type	UTF-8 Encoding	UTF-16 Encoding	UTF-32 Encoding
Z…8Array	0x7FFFFFFFFFFFFFF5	0xBFFFFFFFFFFFFFF5	0x5FFFFFFFFFFFFFF5
Z…16Array	0x3FFFFFFFFFFFFFFA	0x5FFFFFFFFFFFFFFA	0x2FFFFFFFFFFFFFFA
Z…32Array	0x1FFFFFFFFFFFFFFD	0x2FFFFFFFFFFFFFFD	0x17FFFFFFFFFFFFFD
Z…64Array	0x0FFFFFFFFFFFFFFE	0x17FFFFFFFFFFFFFE	0x0BFFFFFFFFFFFFFE
Z…128Array	0x07FFFFFFFFFFFFFF	0x0BFFFFFFFFFFFFFF	0x05FFFFFFFFFFFFFF
ZBCDString	0xFFFFFFFFFFFFFFEA	0xFFFFFFFFFFFFFFFF	0xBFFFFFFFFFFFFFEB

ZDataBlock	0x555555555555554E	0x7FFFFFFFFFFFFFF9	0x3FFFFFFFFFFFFFF9
ZAtomBlock	0xFFFFFFFFFFFFFFEB	0x7FFFFFFFFFFFFFF9	0x3FFFFFFFFFFFFFF9
ZTextArray	0xFFFFFFFFFFFFFFEB	0x7FFFFFFFFFFFFFF9	0x3FFFFFFFFFFFFFF9
ZCharArray	0x7FFFFFFFFFFFFFF5	0xBFFFFFFFFFFFFFF5	0x5FFFFFFFFFFFFFF5

2.16.3 Zero Sized Array Atoms

A zero sized array atom consists of an array header with no following data. It can be used as a placeholder for an array with a non-zero size that is constructed at another time or location. This feature models C and C++ language zero sized arrays, which are allowed as the last data member of a structure or class to define the header of a dynamically sized array.

2.17 Base3z Atom Properties

A Base3z atom can be characterized by data type and encoding properties, including the atom encoding length, encoding format and datum precision, presented in Table 6.

The *fixed encoding length* atom types have starting codons of [0xE000, 0xEC9F] or [0xEE00, 0xEFFF], with lengths of 1, 2, 3, 6 or 11 data codons.

[G20] FixedLengthAtom := NumericAtom | PositionAtom | Enumerated

The *variable encoding length* atom types have starting codons of [0xECA0, 0xEDFF], consisting of all vector format and variable precision atoms.

[G21] VariableLengthAtom := BitString | VPNumericAtom | VPNumericArray | BCDString
PositionArray | NumericArray | DataBlock | AtomBlock | TextArray | CharArray | Symbol

The *scalar encoding format* atom types have starting codons of [0xE000, 0xECA4] or [0xEE00, 0xEFFF], encode a single fixed or variable precision value, with a "size" defined as one.

[G22] ScalarAtom := NumericAtom | PositionAtom | BitString | VPNumericAtom | Enumerated

The *vector encoding format* atom types have starting codons of [0xECA5, 0xEDFF] followed by an Uns32 or Uns64 array size atom, and encode multiple fixed or variable precision values.

[G23] VectorAtom := VPNumericArray | BCDString | PositionArray | NumericArray | DataBlock
| AtomBlock | TextArray | CharArray | Symbol

The *fixed datum precision* atom types have starting codons of [0xE000, 0xECBF] or [0xECA9, 0xEFFF], with single value bit sizes of 8, 16, 32, 64 or 128 for numeric and position types. BCDString atoms have a precision of ~4 bits for each digit or symbol in Table 3.

For completeness, Enumerated and Customized atoms are assigned precisions of 1 bit each. TextString atoms are assigned a precision of 21 bits to account for the 17 Unicode 64K code point planes.

[G24] FixedPrecisionAtom := NumericAtom | PositionAtom | BCDString | PositionArray | NumericArray | DataBlock | AtomBlock | TextArray | CharArray | Symbol | Enumerated

The *variable datum precision* atom types have starting codons of: [0xECA0, 0xECA8], and vary in precision from 32 bits to 131,072 bits in multiples of 32.

[G25] VariablePrecisionAtom := BitString | VPNumericAtom | VPNumericArray

Table 6: Atom Properties

<table>
<tr><th>Atom Type(s)</th><th>Encoding Length</th><th>Encoding Format</th><th>Datum Precision</th></tr>
<tr><td>0xE000
[Uns8, Off64]
0xEC9F</td><td>Fixed</td><td rowspan="3">Scalar</td><td>Fixed</td></tr>
<tr><td>0xECA0
BitString</td><td rowspan="5">Variable</td><td rowspan="3">Variable</td></tr>
<tr><td>0xECA1
[UnsVP, DecVP]
0xECA4</td></tr>
<tr><td>0xECA5
[UnsVPArray,
DecVPArray]
0xECA8</td><td rowspan="3">Vector</td></tr>
<tr><td>0xECA9
BCDString</td><td rowspan="3">Fixed</td></tr>
<tr><td>0xECAA
[Seg16Array, Symbol]
0xEDFF</td></tr>
<tr><td>0xEE00
[Enumerated]
0xEFFF</td><td>Fixed</td><td>Scalar</td></tr>
</table>

Each of these characteristics defines the complete set of Baze3z type tagged atoms.

[G26] Baze3zAtom := (FixedLengthAtom | VariableLengthAtom)
|= (ScalarAtom | VectorAtom)
|= (FixedPrecisionAtom | VariablePrecisionAtom)

The complete atom set includes free text strings:

[G27] Atom := Baze3zAtom | TextString

3 Constant and Binding Atoms

Constant atoms are enumerated atoms that represent encoding, mathematical, physical or application specific constants. Binding atoms are enumerated atoms that prefix or suffix other atoms according to formal sequence rules to form specific atom sequences. A small number of constant and binding atoms are defined in the current version of this document. Additional atoms will be defined in future versions based upon user experience and innovation.

This section provides a formal specification of enumerated atoms using the EBNF style grammar defined in Appendix A. Enumeration rules are indexed with [E] tags. Sequence rules for these atoms are indexed with [S] tags.

[C19] A sequencer or application MUST encode and decode Base3z atom sequences only as specified by the rules defined in this document.

[C20] A sequencer or application MUST use binding atoms in a manner that is consistent with the atom descriptions in this document.

The following definitions provide useful distinctions among atom sequences:

[D7] A **free atom sequence** contains no binding atoms, and therefore only individual atom pattern rules apply to these atom sequences.

[D8] A **primary binding sequence** is defined by the sequence rules for a single binding atom.

[D9] A **complex binding sequence** is defined by the sequence rules for multiple binding atoms or primary sequences.

[D10] A **valid atom sequence** matches all atom pattern and sequence rules defined for all atoms contained in the sequence.

3.1 Boolean Atoms

The first two enumerated atom constants are the Boolean values **true** and **false**. The enumeration and group rules for these constants are:

[E00] ZFalse := U+EE00

[E01] ZTrue := U+EE01

[G28] Boolean := ZFalse | ZTrue

3.2 Null and Void Atoms

The ZNull and ZVoid atoms are interpreted as defined by each application. ZNull often represents a value of zero, while ZVoid often represents the absence of any value or type.

The enumeration rules for these atoms are:

[E02] ZNull := U+EE02

[E03] ZVoid := U+EE03

3.3 Escape Atom

The ZEscape atom prefixes another binding atom to instruct a sequencer or application to ignore the binding of that atom to the adjacent atom sequence. The ZEscape atom has no effect and is invalid when it prefixes non-binding atom types. A ZEscape atom is nullified when prefixed by a second ZEscape atom. Thus, a sequence of 2 or more ZEscape atoms has no effect.

The enumeration rules for these atoms are:

[E04] ZEscape := U+EE04

[S1] EscapeSequence := ZEscape Atom

[C21] A sequencer or application SHOULD report a warning to the application that uses the device or program when invalid or redundant ZEscape atoms are detected.

3.4 Status Sequences

The ZStatus atom prefixes an Uns8, TextArray or AtomBlock atom to form a status indication sequence. Status sequences are used to report device or program status and operational errors “in-band” within an atom output sequence. The value encoded in the Uns8 atom of a StatusCode sequence is one of the numeric constants defined in this section. The content of a StatusMessage or StatusReport sequence is device and program specific.

The atom enumeration rule and sequence rules are:

[E05] ZStatus := U+EE05

[S2] StatusCode := ZStatus Uns8

[S3] StatusMessage := ZStatus TextArray

[S4] StatusReport := ZStatus AtomBlock

[C22] A codec or sequencer SHOULD detect all status conditions and processing errors defined in this specification, and report this information to an application that uses the device or program in a timely and actionable manner.

[C23] A codec, sequencer or application that reports StatusCode atom sequences to other devices or programs MUST do so in a manner consistent with this specification.

3.4.1 General Status Conditions

The status conditions defined here are common to most devices and programs, and can occur during the operations of a codec, sequencer or application of almost any design.

The status constants for these conditions are:

[S5] ZSuccess := U+E000
An operation completed as requested without internal or external errors.

[S6] ZFailure := U+E001
An operation failed as requested without internal errors, indicating a Boolean “false” result.

[S7] ZWarning := U+E002
An operation completed as requested; however, internal or external errors may have occurred that can degrade or halt the device or program operation.

[S8] ZInternalError := U+E003
An operation failed due to an internal error; however, normal operation can continue without restarting the device or program.

[S9] ZCriticalError := U+E004
An operation failed due to an internal error that requires the device or program to be restarted before normal operation can continue.

[C24] A codec or sequencer SHOULD report the error conditions that cause a Critical Error, Internal Error or Warning status condition to an application using the device or program.

3.4.2 Atom Processing Errors

The error conditions defined here are specific to Base3z atom processing, and can occur during the operations of a codec, sequencer or application of any design. These errors indicate the failure of an atom encode, decode or search operation.

The error constant and atom sequence rules for these errors are:

[S10] ZAccessError := U+E005
A storage stream memory access failure was detected.

[S11] ZBitsizeError := U+E006
A variable precision atom with an ill-formed precision tag, or with an incorrect or unsupported bit size value was detected.

[S12] ZBytesError := U+E007
A UTF-16 or UTF-32 storage stream with the wrong byte ordering was detected.

[S13] ZCodonError := U+E008
An ill-formed UTF-8 or UTF-16 code unit sequence was detected.

[S14] ZDataError := U+E009
An expected DataCode codon was absent, or an invalid DataCode codon was detected.

[S15] ZIndexError := U+E00A
A vector formatted atom element index exceeding the vector size was detected.

[S16] ZLengthError := U+E00B
An atom encoding length exceeding the allowed or available storage was detected.

[S17] ZRangeError := U+E00C
An vector formatted atom element range exceeding the vector size was detected.

[S18] ZSizeError := U+E00D
A vector formatted atom with an incorrect or unexpected size value was detected.

[S19] ZSizeDataError := U+E00E
A vector formatted atom with an ill-formed size atom encoding was detected.

[S20] ZSizeLimitError := U+E00F
A vector formatted atom with an unsupported size value was detected.

[S21] ZSizeTypeError := U+E010
A vector formatted atom with an invalid size atom type was detected. The valid size tag atom types are Uns8, Uns32 and Uns64.

[S22] ZTextError := U+E011
An expected TextCode codon was absent, or an invalid TextCode codon was detected.

[S23] ZTypeError := U+E012
An atom of an incorrect or unexpected type was detected.

[S24] ZValueError := U+E013
An incorrect or unexpected atom datum value was detected.

[S25] ZReservedStatusCode := [U+E014, U+E0FF]
This range of Uns8 status codes are reserved for definition in future versions of this document.

[C25] A codec or sequencer SHOULD continue to operate normally without interruption following an occurrence of any of the error conditions defined in this sub-section.

[C26] A codec or sequencer SHOULD report the error conditions defined in this sub-section in the following decreasing priority order:
Access > Bytes > Codon > Length > Data > Text > Type > Size > Index > Range > Value.

3.4.3 Custom Status Codes

The ZStatus atom prefixes an Int32 atom to form a CustomStatusCode sequence. The use of any encoded integer value is defined by a given codec, sequencer or application.

[S26] CustomStatusCode := ZStatus Int32

3.4.4 Error Tags

Error tags replace the starting codon or codons of ill-formed or invalid atoms, indicating the code unit length of the marked atoms. Subsequent operations can identify and skip these atoms with a minimum amount of processing.

The ZAtomError atom replaces an invalid single DataCode atom.

[E06] ZAtomError := U+EE06

The ZErrorSpan atom prefixes an Uns8, Uns32 or Uns64 atom to form an ErrorSpanSequence that replaces the starting 2, 4 or 7 DataCode codons of an ill-formed or invalid atom or atoms. The Unsigned atom encodes the code unit length N of the original atom or atom sequence.

[E07] ZErrorSpan := U+EE07

[S27] ErrorSpanSequence := ZErrorSpan (Uns8 Ux{N-2}) | (Uns32 Ux{N-4}) | (Uns64 Ux{N-7})

3.5 Repeat Sequences

The ZRepeat atom prefixes an Uns8 atom followed by a ScalarAtom or TextCode to form a RepeatSequence. The Unsigned atom encodes a *repetition count* for the subsequent *decoded value*. This atom can be used by applications to run-length encode numeric or text strings.

The enumeration and sequence rules for these atoms are as follows:

[E08] ZRepeat := U+EE08

[S28] RepeatSequence := ZRepeat Uns8 (ScalarAtom | TextCode)

3.6 Array Sequences

3.6.1 Atom Arrays

The ZAtomArray atom prefixes an Uns32 or Uns64 atom followed by a sequence of identical type-size Atom encodings to form an "array of atoms" or ArraySequence. The Unsigned atom encodes the array size as the number of subsequent atoms comprising the array content.

The enumeration and sequence rules for these atoms are as follows:

[E09] ZAtomArray := U+EE09

[S29] ArraySequence := ZAtomArray (Uns32|Uns64) $Atom_{type\text{-}size}\{N\}$

ScalarAtom data types have corresponding packed value VectorAtom types.

3.6.2 Multi-dimensional Arrays

The ZMultiArray atom prefixes an Uns32Array or Uns64Array atom followed by a VectorAtom or ArraySequence encoding to form a MultiArraySequence. The UnsignedArray atom encodes the dimensions set applied to the subsequent VectorAtom or ArraySequence *in row-major order*.

The enumeration and sequence rules for these atoms are as follows:

[E0A] ZMultiArray := U+EE0A

[S30] MultiArraySequence := ZMultiArray (Uns32Array|Uns64Array) (VectorAtom | ArraySequence)

[C27] The product of the dimensions set SHOULD equal the size of the associated VectorAtom or ArraySequence.

[C28] A sequencer or application MUST ignore array indexing requests that exceed the defined dimensions set or size of the associated VectorAtom or ArraySequence, and report an error when such requests occur.

3.7 System Sequences

The ZSystem atom prefixes a TextArray *name* followed by an Atom *value* to form a Base3z encoding SystemSequence. Standard name-value pairs are defined in this document section, and additional application defined name-value pairs are allowed.

The enumeration and sequence rules for this atom are:

[E0B] ZSystem := U+EE0B

[S31] SystemSequence := ZSystem TextArray Atom

[C29] Applications MAY NOT define a SystemSequence TextArray name string that begins with "Base3z," including all letter case variations.

3.7.1 Base3z Specification Version Sequences

A Base3z specification version atom sequence is of the form:

[S32] Base3zVersion := ZSystem TextArray["Base3z.Version"] Atom

3.7.2 Base3z License Grant Sequences

A Base3z license grant atom sequence is of the form:

[S33] Base3zLicense := ZSystem TextArray["Base3z.License"] Atom

3.7.3 Base3z Standard Profile Sequences

A profile sequence encapsulates as set of capabilities supported or required by a given Base3z codec, sequencer or application. Standard profile components specify atom types, array sizes and sequences. Devices and programs can specify custom profile components as required.

A Base3z standard profile atom sequence is of the form:

[S34] Base3zProfile := ZSystem TextArray["Base3z.Profile"] Atom

3.7.4 Base3z Type Format Sequences

Format sequences specify the use of alternate Baze3z data type formatting such as non-IEEE floating point formats in ZFlt… or ZDec… atoms.

A Base3z type format atom sequence is of the form:

[S35] Base3zProfile := ZSystem TextArray["Base3z.Format"] Atom

3.8 Stream Sequences

A ZStream… atom prefixes a TextArray *identifier* followed by an Uns32 or Uns64 atom *length* to form a StreamSequence. The ZStream type atom specifies the encoding form of the subsequent atom stream. The length atom specifies the number of stream code units following that atom, with zero indicating an unknown stream length. The encoding form of a StreamSequence can differ from the encoding form of the subsequent atoms.

The code units following the StreamSequence are memory aligned for the specified stream type by padding the length atom with up to 3 trailing bytes set to zero when “down-sizing” the encoding form.

The enumeration and sequence rules for these atoms are:

[E0C] ZStream8 := U+EE0C
The subsequent atoms are UTF-8 encoded.

[E0D] ZStream16 := U+EE0D
The subsequent atoms are UTF-16 encoded.

[E0E] ZStream32 := U+EE0E
The subsequent atoms are UTF-32 encoded.

[S36] StreamSequence := ZStream8 TextArray (Uns32|Uns64) U8{N}
|= ZStream16 TextArray (Uns32|Uns64) z? U16{N}
|= ZStream32 TextArray (Uns32|Uns64) z? z? z? U32{N}

3.9 Reserved Atoms

Enumerated atoms in this range are reserved for assignment in future versions of this document.

[E0F] ZReserved := U+EE0F

[S37] Reserved := [ZReserved, U+EEFF]

A Notation

The formal grammar of Base3z encoding used in this specification is presented here as a simple Extended Backus-Naur Form (EBNF) style notation. The grammar terminal symbols are the set of all code point values defined in the Unicode® Standard, each represented as:

U+NNNNNN or U+BBBBBBBBBBBBBBBBBBB or U+DDDDDDD

where N is a hexadecimal digit, B is a binary digit, and D is a decimal digit. Leading zeros are insignificant unless noted otherwise, and can be dropped to improve readability. The range of valid values is from 0 to 10FFFF hexadecimal inclusively.

Lower case characters are used as wildcard symbols to define a range of code point values, as in:

U+E0nn or U+11100000bbbbbbbb

which represents the code point range U+E000 to U+E0FF inclusively.

A single code unit of the UTF-8, UTF-16 or UTF-32 encoding forms is indicated as U8, U16 or U32 respectively, and Ux generally. A code point or derived symbol A can be prefixed with U8+, U16+ or U32+ to indicate the corresponding encoding of that symbol, as in: U8+NN or U16+A.

Each rule of the grammar defines a symbol for an expression that matches a sequence of one or more Unicode code points, in the form:

symbol := expression

A defined symbol may be used in a subsequent symbol definition expression.

Alias expressions define symbols beginning with a “Z” prefix that represent specific single code point values, as in:

ZNulCode := U+0000

Range expressions define symbols representing any single code point value within a numeric range inclusively, as in:

[U+0000, U+FFFF]

Sequence expressions define symbols representing a sequence of one or more code point values using the following set of primitive expressions:

A? represents zero or one instance of the symbol A.

A* represents zero or more instances of the symbol A.

A+ represents one or more instances of the symbol A.

A{#} represents an exact number (#) of instances of the symbol A specified by a numeric constant or algebraic formula.

A B represents the symbol A followed by the symbol B.

A|B represents the symbol A or B, but not both.

A-B represents any symbol A that is not symbol B.

(X) represents the (sub)expression X as a single symbol. No other operator precedence rule is defined.

Large or complex rules may be expressed incrementally using the construction form:

```
symbol := expression1
        |= expression2
```

which is equivalent to:

```
symbol := (expression1) | (expression2)
```

B Base3z® Limited Use License

The Base3z Encoding Specification, developer resources, and the technology disclosed in the specification and embodied in the resources are provided by bitLab, LLC under the terms and conditions of this Limited Use License to allow evaluation and public review of this technology.

TERMS AND CONDITIONS (Version 1.2, 15 December 2011)

By using or copying the Base3z specification, developer resources, or items subject to this license, you (the licensee) agree that you have read, understood, and will comply with these terms and conditions.

You may use the information in the Base3z specification and developer resources to design, construct, test and document devices and programs that use the technology disclosed in the specification, and publish these designs and documents, provided that these designs and documents conform to the requirements of the specification, and are accompanied by a notice that any use of these items is subject to the terms and conditions of this license.

You may also acquire, operate or distribute devices and programs that use the technology disclosed in the specification and conform to the requirements of the specification for lawful purposes without fee or royalty, provided that you do NOT sell, lease or trade these devices or programs for money, goods or services, and you do NOT operate these devices or programs in connection with the sale or trade of any goods or services.

All other uses of the technology disclosed in the Base3z specification require the execution of a separate commercial use license agreement with bitLab, LLC. This Limited Use License shall not be construed in any way to limit bitLab, LLC or its successors and assigns from collecting a fee or royalty for these other uses of the technology disclosed in the specification.

BITLAB, LLC PROVIDES THE Base3z SPECIFICATION AND RESOURCES “AS IS,” AND MAKES NO REPRSENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR THAT THE USE OF THE SPECIFICATION OR RESOURCES WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

BITLAB, LLC WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE Base3z SPECIFICATION, RESOURCES OR TECHNOLOGY BY ANY PARTY.

Base3z encoding is protected by U.S. patent 7,982,637 B2. Additional patent applications are pending.

Base3z® is a registered trademark of bitLab, LLC and may NOT be used to advertise or publicize products, services or other activities except those permitted by this license without specific prior written permission. bitLab, LLC claims title to and reserves all rights to the Base3z specification, resources, and the technology disclosed in the specification.

END OF TERMS AND CONDITIONS

C Technical Notes

C.1 Base3z Design Considerations

C.1.1 Total Bit Utilization

The Base3z fixed precision numeric atom set encodes 16 binary data types with a total of 944 data bits using 1376 encoding bits. When type tags for the atoms are counted, an additional 4 bits per type or 64 bits of information is encoded for the group. The total encoding efficiency of this set is (64+944)/1376 or 73.26 percent using UTF-16 codons, and assuming an even usage distribution of the individual types. Specific atom bit-size efficiencies are as follows:

8: (4+8)/16 = 75%, 16: (4+16)/32 = 63%, 32: (4+32)/48 = 75%, 64: (4+64)/96 = 71% 128: (4+128)/176 = 75%

The high 0xE nibble of each Base3z UTF-16 data codon serves as a bit-error detection field. These bits are therefore not lost as overhead, raising the total bit utilization by this measure to over 98% for a UTF-16 encoding of this atom group.

A UTF-8 encoding of the same Base3z fixed precision numeric atom set requires 50% more storage than a UTF-16 encoding, yielding a spatial efficiency of 48.84 percent. However, each UTF-8 encoded DataCode codon is 3 bytes of the form 0xEE, 0x10xxxxxx, 0x10xxxxxx, which fixes 12 bits of each 24 bit codon for bit-error detection, again raising the total bit utilization to over 98 percent.

Fixed precision array atom encodings have a total bit utilization of 83% for the smallest single element (3-codon) arrays, at least 90% for atoms of 10 codons in length, and at least 95% for atoms of 20 or more codons in length.

C.1.2 Fixed Length vs. Variable Length Encoding

Variable length byte-wise data encoding using a single continuation-bit per byte provides a storage efficiency limit equal to 7 of 8 bits, or 87.5 percent. However, the set of well-formed Unicode UTF-8 code unit sequences does not include all possible byte sequences, forcing a UTF-8 based variable length encoding scheme to use the 7-bit code unit range U+00 to U+7F. The storage efficiency limit is thus reduced to 6 of 8 bits, or 75 percent.

The Base3z encoding length of a single numeric value is a constant for each data type, while the length of a numeric array is a constant plus a simple function of the data type and array size. These properties enable extremely fast storage calculations for numeric atoms, *and indexed access to individual array elements*. Simple variable length encoding methods do not allow these features.

Integers, real numbers, bit-strings and opaque binary data are Base3z encoded as a *variable length sequence* of 12-bit data payloads in UTF-8 *or* UTF-16 E-block codons. This multiple encoding scheme capability is a key feature of Base3z encoding.

None of the common processor word bit-sizes is a multiple of 6 or 7 bits. Decoding a variable length byte sequence to a register can require overflow detection and error handling code. Base3z encoding using UTF-16 codons is faster than base 64 or base 128 variable length integer methods.

C.1.3 Type-Size-Values vs. Type-Length-Values Array Encoding

The encoding length of a Base3z vector formatted type is derived from the type and encoded size using a simple formula specific to the type datum precision. For example, the UTF-16 length of a 50 element Uns32Array would be:

1 (type) + 1 (size) + (50 elements * 8 nibbles/element + 2) / (3 nibbles/codon) = 136 data codons

Decoding the array size and computing the encoding length require similar numbers of operations to compute for the case of an Uns8 array size atom. Decoding an Uns32 or Uns64 size atom requires about 3 or 6 times this number of operations.

The length calculation time could be avoided when scanning past the atom if the length were encoded in place of the array size. Searching for a particular array type *and* size could be accomplished by using the *expected* length of the array atom based upon the type and size. However, this approach presents a new potential error condition when *the atom length is not the exact length of any array size*.

For a given array type and size, exactly one data length is produced by the corresponding formula, which is the length that a codec is required to encode or decode. For a given array type and length; however, the maximum number of elements (size) that can be encoded can leave unused codons at the end of the atom. There are 11 possible lengths for an Uns128Array of any given size.

Verification of array atom encodings in this alternative scheme would require detection of non-minimal encoding lengths, and possibly scrubbing the excess codons. Moreover, the required encoding length of a given array size must be computed when allocating storage for arrays. All things considered, the total processing burden of this approach likely exceeds that of the current Base3z design.

C.1.4 Private-Use Areas: Plane Zero vs. Planes F and 10

The Base3z method uses a 4096 private-use code point block starting at U+E000 in the Basic Multilingual Plane (zero) to encode binary data in 12-bit units. The limiting storage efficiency of this encoding depends upon the Unicode encoding form used to represent these code points as follows:

UTF-8: 12 data bits / (3 * 8-bit code units) = 50%
UTF-16: 12 data bits / (1 * 16-bit code unit) = 75%
UTF-32: 12 data bits / (1 * 32-bit code unit) = 37.5%

Single data values with power of two bit-sizes leave either 4 or 8 encoding bits unused for data that are used to encode the value type. Vector type and size information is encoded as either 12 or 24 bits to complete a data type system that, except for the trailing element in 2 of every 3 vectors, achieves otherwise perfect bit utilization.

An alternate encoding method can be designed using either of the Unicode private-use areas starting at U+F0000 or at U+100000, planes F and 10. The optimal use of one of these 64-K code point blocks is to encode 16-bits of binary data as a member of the block, yielding a limiting efficiency for each Unicode encoding form as follows:

UTF-8: 16 data bits / (4 * 8-bit code units) = 50%
UTF-16: 16 data bits / (2 * 16-bit code units) = 50%
UTF-32: 16 data bits / (1 * 32-bit code unit) = 50%

However, this design is deficient for several reasons that preclude its use by the Base3z method:

1. The maximum efficiency is only 50%, compared with 75% for Base3z UTF-16 encoding.
2. Adding type information further reduces the encoding efficiency for single value data types.
3. Using the last 2 code points of these blocks for inter-application information transfers is generally forbidden by the Unicode® specifications.

C.1.5 Alternate Encoding Blocks in the BMP Private-Use Area

The 4096 code point block used for Base3z encoding could be located at any starting code point from U+E000 to U+E900. This start value would be *added* to each 12-bits of data during encode operations and *subtracted* during decode operations. The performance of software codecs would be the same using this method as the OR/AND method used with the U+E000 starting value.

A hardware codec can be implemented for the U+E000 block by simply shifting data bits and inserting or deleting the bit pattern “1110” during register save and load operations without the additional gates for arithmetic operations required with other block locations. The effective processing overhead of Base3z encoding approaches zero when encoding and decoding takes place during store and load operations, making the minimum hardware solution a compelling design choice.

The use of alternate encoding blocks will require applications to exchange the block location at the atom sequence or application session level in a standardized manner to be generally useful, leaving U+E000 as the default block location. Such a feature may be included in a future version of the Base3z specification if needed.

C.1.6 UTF-8 vs. UTF-16 Encoding

The relative merits of UTF-8 and UTF-16 encoding of *text characters* have been debated since the introduction of UTF-8 in 1993 as a means to allow existing byte-oriented I/O systems and string processing libraries to handle the Unicode character database with little or no modification.

Base3z defines text and data atoms as Unicode *code point* sequences. The set of all valid Base3z atom encodings, excluding AtomBlock atoms, is identical for the UTF-8 and UTF-16 encoding forms. However, the encoding of data as U+Exxx code points requires these additional considerations in the selection of a Unicode encoding form:

1. The distribution of code points to be processed, stored or transferred by an application will be shifted toward the U+E000 to U+EFFF code block as more Base3z atoms are used. A UTF-8 encoding requires 3 bytes for these code points, while UTF-16 requires only 2 bytes.
2. The speed of Base3z data encode and decode operations is significantly higher using UTF-16 code units; generally, more than double the UTF-8 performance.
3. Fully optimized Base3z codecs are somewhat easier to code for UTF-16 encodings.
4. Conversion between UTF-8 and UTF-16 streams can be very fast.

Many Base3z application designs can be optimized by using UTF-16 encodings internally while supporting UTF-8 based external interfaces when necessary using conversion.

C.1.7 Base3z Data Transfer as Base64

Unicode code unit streams (UTF-8 or UTF-16) require an 8-bit clean transport pathway. Some legacy protocols provide only 7-bit pathways, for which transfer encodings such as Base64 provide a method of transporting binary data as “printable only” USASCII characters. This same processing solution can be applied to Base3z atoms.

A Base3z DataCode codon can be converted to a pair of Base64 characters containing the 12 data bits of the codon, *assuming the constant 0xE upper nibble*, with no increase in storage size and very little processing overhead. For example, an Uns32 atom with a hex value of 0x12345678 encodes as three 16-bit words 0xE212 0xE345 0xE678 using UTF-16, which converts (in big-endian order) to the Base64 character string “ISNFZ4.”

TextCode codons require 4 Base64 characters for every 3 bytes of UTF-8 or UTF-16 storage conversion, which incurs a 33% storage overhead for TextArray atom content and TextString (free text) atoms. In addition, applications must mark the boundaries of TextString atoms by some means if they are mixed with other atoms within the same Base64 character string.

A direct Base64 encoding of tagged data types is certainly possible and useful. While providing similar storage efficiency and some added transportability, this approach has several disadvantages compared to Base3z encoding:

1. Base3z encoding and decoding is up to twice as fast (using UTF-16) as Base64 encoding.
2. Base3z encoded data can be scanned for a specific atom tag without decoding, while Base64 encoded data requires partial decoding to reveal atom tags for subsequent pattern matches.
3. Base3z encoded data provides for a modest level of error detection due to the 0xE constant upper nibbles, while Base64 encoded data provides essentially no inherent error detection.
4. Base3z encoded data is distinct by code point range within a Unicode stream, while Base64 encoded data requires additional delimiting character storage and context processing.

The discontiguous and disorderly Base64 mapping of 6-bit data chunks to USASCII characters is the source of the performance disadvantages in items 1 and 2. Fast Base64 encoding and decoding utilize both table lookups and logical (dis) assembly of binary values, while Base3z operations require only logical (dis) assembly of binary values.

C.2 Transfer Encoding Performance

Encoding large sections of memory as text for transfer among information devices provides a useful test of the performance limits of Base3z encoding. Test results for Base16 (binhex), Base64 encoding [**IETF RFC 4648**] and Base3z encoding are presented in Table 7. Results for a memcpy() operation are included as a baseline for the tests. These tests use ideal length data sets without error checking in order to determine the maximum potential speed of each encoding method.

Table 7: Transfer Encoding Performance

Input Stream	Encoding Modulus	Output Stream	Decoded Bytes	Encoded Bytes	Storage Ratio	Encode nSec/byte	Decode nSec/byte
bytes	Base16	ASCII	262144	524288	50.0%	1.721	1.731
bytes	Base64	ASCII	262143	349524	75.0%	1.617	1.475
bytes	Base3z	UTF-8	262143	524286	50.0%	1.615	1.380
words	Base64	ASCII	262143	349524	75.0%	0.988	1.043
bytes	Base3z	UTF-16	262143	349524	75.0%	1.029	0.871
quads	Base3z	UTF-16	262143	349520	75.0%	0.548	0.478
void	memcpy	void	262144	262144	100.0%	0.128	0.128

The test routines were implemented in C++ and compiled for maximum speed. Each test encodes 256 Kbytes of random values from memory as a text string to memory, and then decodes the string back to the original data. A notebook computer with a 2.4 GHz Intel Core 2 Duo processor, 3MB L2 Cache and 2Gbytes of 1067 MHz of DDR3 memory was used for these tests.

Execution times for each test encode and decode steps were measured using a high-resolution counter based upon the processor clock. The test data size was chosen to allow the entire data set to reside in the L2 cache, and the tests to execute within a single OS time slice. The tests were repeated to discover the best times for each. The implementation details of each test are discussed below.

Bytes to Base16 as ASCII

The encode operation converts each test byte value to an ASCII hex digit pair from “00” to “FF” using a 256 entry LUT containing all such pairs:

```
uns32 j = V << 1;
E[0] = encode16[j];
E[1] = encode16[j+1];
```

The decode operation uses a 256 entry reverse LUT containing the values 0x00 to 0x0F as the entries for hex digits ‘0’ to ‘9’ and ‘A’ to ‘F’ to compute the original byte value as:

```
D = (decode16[E[0]] << 4) + decode16[E[1]];
```

The encoding requires 2 bytes of storage per test byte, yielding a storage efficiency of 50%.

Bytes to Base64 as ASCII

The encode operation converts every 3 test bytes to a sequence of 4 ASCII base64 “digits” from the ordered set:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

Every 6 bits of the test byte sequence are mapped in big-endian order to a base64 digit using a 64 entry LUT containing all such pairs:

```
uns32 v = (V[0] << 16) | (V[1] << 8)  | V[2];
E[0] = encode64[v >> 18];
E[1] = encode64[(v >> 12) & 0x3F];
E[2] = encode64[(v >> 6) & 0x3F];
E[3] = encode64[v >> & 0x3F];
```

The decode operation uses a 256 entry reverse LUT containing the values 0x00 to 0x3F as the entries for the base64 digits to compute the original byte values:

```
D[0] = (decode64[E[0]] << 2) | (decode64[E[1]] >> 4);
D[1] = (decode64[E[1]] << 4) | (decode64[E[2]] >> 2);
D[2] = (decode64[E[2]] << 6) | decode64[E[3]];
```

The encoding requires 4 bytes of storage for 3 test bytes, yielding a storage efficiency of 75%.

Bytes to Base3z as UTF-8

The encode operation converts every 3 test bytes to a pair of Base3z UTF-8 DataCode codons:

```
11101110, 10bbbbbb, 10bbbbbb, 11101110, 10bbbbbb, 10bbbbbb
```


Every 3 nibbles of the test byte sequence are mapped in big-endian order to a Base3z DataCode codon by formula:

```
uns32 v = (V[0] << 16) | (V[1] << 8)  | V[2];
E[0] = 0x0EE;
E[1] = 0x080 | (v >> 18);
E[2] = 0x080 | ((v >> 12) & 0x3F);
E[3] = 0x0EE;
E[4] = 0x080 | ((v >> 12) & 0x3F);
E[5] = 0x080 | (v & 0x03F);
```

The decode operation computes the original byte values by formula:

```
D[0] = (E[1] << 2) | ((E[2] & 0x03F) >> 4);
D[1] = (E[2] << 4) | ((E[4] & 0x03F) >> 2);
D[2] = (E[4] << 6) | (E[5] & 0x03F);
```

The encoding requires 6 bytes of storage for 3 test bytes, yielding a storage efficiency of 50%.

Words to Base64 as ASCII

The encode operation converts every 3 test bytes to a sequence of 4 ASCII base64 “digits” from the ordered set:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

Every 3 nibbles of the test byte sequence are mapped in big-endian order to a base64 *digit pair* using a 4096 entry LUT containing all such pairs:

```
uns16* e = (uns16*)E;
e[0] = ((uns16*)encode64)[((V[0] << 4) | (V[1] >> 4))];
e[1] = ((uns16*)encode64)[(((V[1] << 8) | V[2]) & 0x0FFF)];
```

The decode operation uses a 32K entry reverse LUT containing the values 0x000 to 0xFFF as the entries for the base64 *digit pairs* to compute the original byte values:

```
uns16* e = (uns16*)E;
D[0] = decode4K[e[0]] >> 4;
D[1] = ((decode4K[e[0]] & 0x0F) << 4) | (decode4K[e[1]] >> 8);
D[2] = (uns8)decode4K[e[1]];
```

The encoding requires 4 bytes of storage for 3 test bytes, yielding a storage efficiency of 75%.

Bytes to Base3z as UTF-16

The encode operation converts every 3 test bytes to a pair of Base3z UTF-16 DataCode codons:

```
0xExxx, 0xExxx
```

Every 3 nibbles of the test byte sequence are mapped in big-endian order to a Base3z DataCode codon by formula:

```
E[0] = 0x0E000 | ((uns16)V[0] << 4) | (V[1] >> 4);
E[1] = 0x0E000 | ((uns16)(V[1] & 0x0F) << 8) | V[2];
```

The decode operation computes the original byte values by formula:


```
D[0] = (uns8)(E[0] >> 4);
D[1] = (uns8)(E[0] << 4) | ((E[1] >> 8) & 0x0F);
D[2] = (uns8)E[1];
```

The encoding requires 4 bytes of storage for 3 test bytes, yielding a storage efficiency of 75%.

Quads to Base3z as UTF-16

The encode operation converts every 12 test bytes 4 at a time to a sequence of 8 Base3z UTF-16 DataCode codons:

```
0xExxx, 0xExxx, 0xExxx, 0xExxx, 0xExxx, 0xExxx, 0xExxx, 0xExxx
```

Every 3 nibbles of the test byte sequence are mapped in big-endian order to a Base3z DataCode codon by formula:

```
uns32* v = (uns32*)V;
E[0] = 0x0E000 | (v[0] >> 20);
E[1] = 0x0E000 | ((v[0] >> 8) & 0x0FFF);
E[2] = 0x0E000 | ((v[0] << 4) & 0x0FF0) | (v[1] >> 28);
E[3] = 0x0E000 | ((v[1] >> 16) & 0x0FFF);
E[4] = 0x0E000 | ((v[1] >> 4) & 0x0FFF);
E[5] = 0x0E000 | ((v[1] << 8) & 0x0FF0) | (v[2] >> 24);
E[6] = 0x0E000 | ((v[2] >> 12) & 0x0FFF);
E[7] = 0x0E000 | (v[2] & 0x0FFF);
```

The decode operation computes the original byte values 4 at a time by formula:

```
uns32* d = (uns32*)D;
d[0] = ((uns32)e[0] << 20) | ((0x0FFF & (uns32)e[1]) << 8) |
       ((0x0FF0 & (uns32)e[2]) >> 4);
d[1] = ((uns32)e[2] << 28) | ((0x0FFF & (uns32)e[3]) << 16) |
       ((0x0FFF & (uns32)e[4]) << 4) | ((0x0F00 & (uns32)e[5]) >> 8);
d[2] = ((uns32)e[5] << 24) | ((0x0FFF & (uns32)e[6]) << 12) |
       (0x0FFF & (uns32)e[7]);
```

The encoding requires 4 bytes of storage for 3 test bytes, yielding a storage efficiency of 75%.

C.3 Protocol Examples

C.3.1 Block Structured Data

Any memory range within a process address space can be encoded as DataBlock atom content at a rate near 2 GBytes/second using a fast modern processor core. This minimal application of Base3z encoding produces a text formatted data block that fully conforms to the Unicode® Standard. However, the data encoding is device and program specific, with almost zero intrinsic information value beyond statistical patterns. From a small data blob to a full “core dump” image, this encoded data requires external schema and entry maps to provide structure and meaning to the atom content.

```
struct Patient                  DataBlock[100]
{ chr16     surName[32];        {  Exxx, Exxx, Exxx, …
  chr16     givenName[32];        Exxx, Exxx, Exxx, …
```



```
  uns16     birthDate[3];           Exxx, Exxx, Exxx, …
  flt32     heightCM;               Exxx, Exxx, Exxx, …
  flt32     weightKG;               Exxx, Exxx, Exxx, …
  Patient*  lastPatient;            Exxx, Exxx, Exxx, …
  Patient*  nextPatient;         }
};
```

C.3.1.1 Tagged Data Sequences

A large amount of type information is added to a Base3z encoding when each datum within a memory range is encoded as a distinct atom. These type-tagged atom sequences provide patterns to search for specific atoms and validate atom decode operations. Moreover, these encodings are both device and program independent, as data alignment and byte ordering are normalized. The additional storage require for this form of encoding is negligible overall, while the encoding and decoding rates are at least half the rate of a single DataBlock atom.

```
struct Patient
{ chr16     surName[32];            TextArray[32];
  chr16     givenName[32];          TextArray[32];
  uns16     birthDate[3];           Uns16Array[3];
  flt32     heightCM;               Flt32;
  flt32     weightKG;               Flt32;
  Patient*  lastPatient;            Ptr32;
  Patient*  nextPatient;            Ptr32;
};
```

Programs written in C-style languages make extensive use of pointers to implement data references and relationships. Storing and retrieving data that contains pointers is the fundamental problem solved by object database technology. A memory pointer or offset can be Base3z encoded and decoded while “swizzling” the value to reference the same logical object in either the encoding data stream or the decoded memory space. The encoded pointer or offset type is “void*” by default. The effective type is defined by the referenced atom type, or explicitly tagged according to an associated schema.

C.3.1.2 Structured Data Blocks

Statically typed languages do not require embedded type information unless reflective operations are also supported. The absence of type tags allows a useful feature of C-language data structuring; that is, the address of the first element of an array or structure is the address of the array or structure itself.

Encoding structure boundaries using AtomBlock atoms; however, can enhance search performance, and provides additional pattern information for type discovery and data validation. AtomBlock atoms can be assigned an application type using Customized atoms and other identification techniques.

```
#define ZPatient ZCustomized+1

struct Patient                  ZPatient AtomBlock[92]
```



```
{ chr16     surName[32];          {  TextArray[32]
  chr16     givenName[32];           TextArray[32]
  uns16     birthDate[3];            Uns16Array[3]
  flt32     heightCM;                Flt32
  flt32     weightKG;                Flt32
  Patient*  lastPatient;             Ptr32
  Patient*  nextPatient;             Ptr32
};                                   }
```

C-style data, including arrays, structures and unions can be encoded using only scalar Base3z atoms. This approach does not benefit from packed array data encoding; however, and requires additional application code to manage arrays.

C.3.1.3 Identifiers and Schemas

A complete encoding of an application data structure or object may include type or class identifiers and member or instance identifiers. Type identifiers can be created using Customized atoms alone or as prefixes to other atoms.

Applications can create structure type definitions using a Base3z encoding pattern that parallels the encoding of a type instance using atom type constants, array size atoms and Symbol atoms:

```
struct Patient                  ZStruct ZPatient AtomBlock[60]
{ chr16     surName[32];        {  ZTextArray Uns32=32 ZSymbol:07 "surName"
  chr16     givenName[32];        ZTextArray Uns32=32 ZSymbol:09 "givenName"
  uns16     birthDate[3];         ZUns16Array Uns32=3 ZSymbol:09 "birthDate"
  flt32     heightCM;             ZFlt32 ZSymbol:08 "heightCM"
  flt32     weightKG;             ZFlt32 ZSymbol:08 "weightKG"
  Patient*  lastPatient;          ZPatient ZPtr32 ZSymbol:0B "lastPatient"
  Patient*  nextPatient;          ZPatient ZPtr32 ZSymbol:0B "nextPatient"
};                              }  ZSymbol:07 "Patient"
```

D References

Normative:

IEEE STD 754

Institute of Electrical and Electronic Engineers, Inc. *IEEE Std 754-2008. The IEEE Standard for Floating-Point Arithmetic.* (New York, NY, IEEE, 2008) See http://standards.ieee.org for access to this document online.

IETF RFC 2119

Internet Engineering Task Force. *RFC 2119: Key words for use in RFC's to Indicate Requirement Levels.* Scott Bradner, 1997. See http://www.ietf.org/rfc/rfc2119.txt

Unicode

The Unicode Consortium. The Unicode Standard, Version 5.0, defined by: *The Unicode Standard 5.0* (Boston, MA: Addison-Wesley, 2007. ISBN 0-321-48091-0) Online versions are available at: http://www.unicode.org/unicode/standard/versions

Informative:

IETF RFC 4648

The Internet Society. *RFC 4648: The Base16, Base32, and Base64 Data Encodings.* S. Josefson, 2006. See http://tools.ietf.org/rfc/rfc4648.txt

W3C XML 1.0

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, editors. *Extensible Markup Language (XML) 1.0 Fifth Edition*. World Wide Web Consortium, 2008. See http://www.w3.org/TR/xml/