Digital AV SDK – Record layouts & File inventory

Release Version: Omega-3.5

VERSION IDENTIFIERS
Digital-AV SDK: Ω35$_7$
SDK Document: Ω39$_1$

Digital-AV Software Development Kit provides the foundation for a fully working bible application, with no external dependencies. The SDK provides everything, including data with indexes. Be up and running in under an hour! Easily jumpstart your development by leveraging sources in the foundational releases.

Directory Content (48 bytes per record)

Content Label char[16]	Content Offset uint32	Content Length uint32	Record Length uint32	Record Count uint32	Content Hash uint64[2]
Directory	0	384	48	8	00000000000000000000000000003507
Book	384	3,216	48	67	BE0DE7885CDEE6C1ABAEF028E0A9F967
Chapter	3,600	7,134	6	1,189	29D5C0D1AFF79C95BE3A964891126FBE
Written	10,734	18,951,624	24	789,651	A1F54F560E73511DA77DCDB91A85EFDB
Lexicon	18,962,358	246,258	0	12,567	F1C7694D3C5B15A526845D7A4946BDFE
Lemmata	19,208,616	182,344	0	15,171	B64F907ABC54470F2D227D8AC5703E33
OOV-Lemmata	19,390,960	7,754	0	771	ADEA45027082EC56EA59B079EF94C96F
Names	19,398,714	60,727	0	2,470	B7885CB9C8F0293A3845818BD5A4DCEC

The Digital-AV SDK (AV SDK) is entirely file based. There are zero dependencies and zero language bias (all programming languages can read a file). The Ω-Series SDK is derived from the earlier Z-Series SDK. The main difference is the Ω-Series use a single content file, beginning with a content directory as depicted above (The Z-Series releases have multiples files and a separate inventory file similar to the directory). The directory identifies the content sections for easy deserialization. The first field is the label, followed by an offset, and a length (in bytes). It also includes record length: zero (0) indicates that the record is variable length; non-zero length means that the record is fixed length. Record count, and content hash [using MD5] are the final fields for each directory item.

The file format defined in this document pertains to the Ω-Series releases. Both Ω and Z series formats are similar, but not identical. Consequently, some formats that have been revised in the Ω-Series do not have corresponding revisions in the Z-Series SDK specification.

Written Content (24 bytes per record)

Record # *0 bits*	Hebrew | Greek *4 x uint16*	B:C:V:W *4 x byte*	Caps *2 bits*	Word Key *14 bits*	Punc *byte*	Transition *byte*	PN+POS(12) *uint16*	POS(32) *uint32*	Lemma *uint16*
0	0x391C 0 0 0	1:1:1:10	0x8___	0x0015 (in)	0x00	0xE8	0x00E0	0x40080470	0x0015
1	0x391C 0 0 0	1:1:1:9	0x0___	0x0136 (the)	0x00	0x00	0x0D00	0x00000094	0x0136
2	0x391C 0 0 0	1:1:1:8	0x0___	0x24F9 (beginning)	0x00	0x00	0x4010	0x000001DC	0x24F9
…	<< Beginning of Genesis 1 depicted above >>								
0xBDDB9	0x25A0 0 0 0	66:22:21:35	0x8___	0x0136 (the)	0x00	0xE0	0x0D00	0x00000094	0x0136
0xBDDBA	0x25A0 0 0 0	66:22:21:34	0x8___	0x2CB2 (revelation)	0x00	0x00	0x4010	0x000001DC	0x2CB2
0xBDDBB	0x0978 0 0 0	66:22:21:33	0x0___	0x001D (of)	0x00	0x00	0x0400	0x80004206	0x001D
…	<< Beginning of Revelation 1 depicted above >>								
0xC0C91	0x1460 0 0 0	66:22:21:3	0x0___	0x015C (you)	0x00	0x00	0x20C0	0x00083BBD	0x015C
0xC0C92	0x0F74 0 0 0	66:22:21:2	0x0___	0x0036 (all)	0xE0	0x04	0x0D00	0x00000004	0x0036
0xC0C93	0x0119 0 0 0	66:22:21:1	0x8___	0x018A (amen)	0xE0	0xFC	0x8000	0x8000550E	0x018A
<< End of Revelation 22:21 depicted above >>									

The most substantial content contained in the directory is that which is Written. It represents the stream of words for each verse of each chapter of each book of the KJV bible. These are not text files. Therefore, they are quite compact. Several fields are index lookups into other SDK content. Collectively, the entire content manifests an efficient database of word embeddings that can compactly reside in RAM.

The first field of Written content contains Strong's numbers[1]. These are a numeric representation of the original Hebrew/Greek words from which the sacred text was originally translated.

Hebrew | Greek

Strong's #1	Strong's #2	Strong's #3	Strong's #4
1st numeric	2nd numeric	3rd numeric	4th Strongs #

[1] Refer to the Strong's Exhaustive Concordance for additional backround information. The Digital-AV has, at most, four Strong's numbers per English word in the Old Testament. By contrast, there are at most, three Strong's numbers per English word. To maintain a fixed length record format, four slots allotted.

The Ω-series releases eliminate the AV-Verse index and place that information directly in the Written content records instead. Indexing into Written from Chapter provides verse coordinates and word counts. Navigating to subsequent verses is accomplished via the word-count for the verse. The first word always contains the word count of the verse; each subsequent word contains a countdown until one (i.e. that last word of the verse is marked with a *:*:*:1)

Word Key & Capitalization Field

Description	Bit Pattern (Hex)
English Word	0x3FFF (mask for lexicon lookup)
1st Letter Cap	0x8000 (example: Lord)
All Letters	0x4000 (example: LORD)

The next sixteen bits can be thought of as two distinct fields: the first 2 bits, ***Caps***, identify whether to apply capitolization rules to the lexical word. 0x8___ means to capitolize the first letter of the word (e.g. Lord). 0x4___ means to capitolize all letters of the the word (e.g. LORD). Clearly, in English, the first letter of the first word of a sentence is capitolized, and these bits facilitate all such capitolization rules. When no bits are set, this indicates that the word should be represented exactly as it appears in the lexicon. The remaining 14-bits are referred to as the ***Word Key*** (a lookup key into the Lexicon). The next field is the ***Punctuation*** byte. Each word can be preceded by punctuation (e.g. an open parenthesis). More often, punctuation follows the word. The ***Punctuation*** byte also contains possible ***Decoration***. Decoration includes italisized words, and words spoken by Jesus, which some bibles represent as red-colored text. The field is entirely bitwise and many forms of punctuation and decoration can simultaneouslt apply to a single word in the text.

Punctuation & Decoration

Description	Bits
PUNC::clause	0xE0
PUNC::exclamatory	0x80
PUNC::interrogative	0xC0
PUNC::declarative	0xE0
PUNC::dash	0xA0
PUNC::semicolon	0x20
PUNC::comma	0x40
PUNC::colon	0x60
PUNC::possessive	0x10
PUNC::closeParen	0x0C
MODE::parenthetical	0x04
MODE::italics	0x02
MODE::Jesus	0x01

Transition bits are a composition of Verse-Transitions and Segment-Markers. These represent a compact mechanism for data file traversal, obviating the need for leveraging additional index files. The five left-most bits mark book, chapter, and verse transitions. The three right-most bits mark linguistic segmentation [sentence and/or phrase] boundaries. Tthese boundaries are based upon a verse transitions and punctuation.

Verse Transitions

Description	5-bits
EndBit	0x10
BeginningOfVerse	0x20
EndOfVerse	0x30
BeginningOfChapter	0x60
EndOfChapter	0x70
BeginningOfBook	0xE0
EndOfBook	0xF0
BeginningOfBible	0xE8
EndOfBible	0xF8

Segment Transitions

Description	3-bits
HardSegmentEnd	0x04
CoreSegmentEnd	0x02
SoftSegmentEnd	0x01
RealSegmentEnd	0x06

Hard Segments: . ? !
Core Segments: :
Real Segments: . ? ! :
Soft Segments: , ; ( ) --

PN+POS(12) is a sixteen bit field with the left-most nibble representing Person Number (PN). PN applies to pronouns and verb casing. Early Modern English was richer than our English today, with additional pronouns and verb cases for Second-Person-Singular and Third-Person-Singular. The Digital-AV captures and preserves all such case markings. For instance, ***thy*** is second-person singular whereas Early Modern English ***you*** is always plural form of this pronoun. The SDK encodes the markings for both person and number using the binary representation depicted in the table to the right. Similarly, the remaining twelve bits provide course part-of-speech markers.

Person/Number (4 bits)

Description	Left-Most Nibble
Person bits	0x3--- (0b--11)
Number bits	0xC--- (0b11--)
Indefinite	0x0--- (0b--00)
1st Person	0x1--- (0b--01)
2nd Person	0x2--- (0b--10)
3rd Person	0x3--- (0b--11)
Singular	0x4--- (0b01--)
Plural	0x8--- (0b10--)
WH*	0xC--- (0b00--)

POS (12 bits)

NounOrPronoun	0x-03-
Noun	0x-01-
Noun: unknown gender	0x-010
Proper Noun	0x-03-
Pronoun	0x-02-
Pronoun: Neuter	0x-021
Pronoun: Masculine	0x-022
Pronoun: Non-feminine*	0x-023
Pronoun: Feminine	0x-024
Pronoun/Noun: Genitive	0x-0-8
Pronoun: Nominative	0x-06-
Pronoun: Objective	0x-0A-
Pronoun: Reflexive	0x-0E-
Pronoun: no case/gender	0x-020
Verb	0x-1--
to	0x-200
Preposition	0x-400
Interjection	0x-800
Adjective	0x-A00
Numeric	0x-B00
Conjunction	0x-C0-
Determiner	0x-D0-
Particle	0x-E00
Adverb	0x-F00

The remaining twelve bits of POS(12) provide bitwise information on the word usage in the context of this verse. The table to the left shows the meaning of the various bits. There is an additional POS(32) field that has much greater fidelity on the part-of-speech for the word. POS(32) is a five-bit encoding of a human readable string. See the section labeled "Additional notes about Part-of-Speech in Digital-AV" for additional details.

* ***his*** is used ambiguously in the Authorized Version for third-person-singular pronouns. ***his*** is either masculine or neuter (***its*** appears just once in the sacred text). Therefore, ***his*** can neither be uniformly marked as masculine, nor neuter. Instead, we mark the genitive pronoun ***his*** as non-feminine.

Book content provides indicies into Chapter content, Written content. It also provides chapter-counts and word-counts (for each of the sixty-six books of the bible). It reserves a fixed sixteen-byte field for the book-name, a fixed nine-byte field (2+3+4) for 2-character, 3-character, and 4-character abbreviations. The remaining nine bytes are a comma-delimited list of any additional alternate abbreviations.

Book Content (48 bytes)

Book Number *byte*	Chapter Count *byte*	Chapter Index *uint16*	Writ Count *uint16*	Writ Index *uint32*	Book Name *16 bytes* (utf8)	Abbreviations (utf8) a2 a3 a4 *(12 bytes)*	Alternates *(10 bytes)*
0	0	0	0	0x3507	Omega 3.5.07----	35-o35-Ω35 -	----------
1	50	0	38262	0	Genesis---------	Ge-Gen-Gen--	Gn--------
2	40	50	32685	38262	Exodus----------	Ex-Exo-Exod-	----------
3	27	90	24541	70947	Leviticus-------	Le-Lev-Lev--	Lv--------
66	22	1167	11995	777656	Revelation------	Re-Rev-Rev--	

The dashes (-) represent zero ('\0'). The nine byte field above, namely "a2 a3 a4" comprises 2-character, 3-character, and 4-character abbreviations. AV-Book.ix has an updated format in the Z32 release. Note that the newer format now contains 67 records instead of 66. The zeroth record contains metadata about the revision and makes record #1 correspond to book #1.

Chapter Content (6 bytes)

Record # 0 bits	Writ Index Uint16	Writ Count uint16	Book Num byte	Verse Count byte
0x000 (genesis:1)	0	797	1	31
0x001 (genesis:2)	797	632	1	25
0x002 (genesis:3)	1429	695	1	24
	. . .			
0x4A2 (revelation:20)	10196	477	66	15
0x4A3 (revelation:21)	10673	749	66	27
0x4A4 (revelation:22)	11422	573	66	21

NOTE:
Chapter content differs significantly from earlier revisions, as it how includes book number and verse count, superseding the Verse-Index found in the Z-Series releases. Verse look-up is now performed using the WritIndex and referencing the B:C:V:W field of Written content. As WritIndex is now 16-bits, it needs to be added to Book[num]. WritIndex on implementations where deserialization of Written content instantiates a single array (It is recommended that deserialization creates 66 distinct Written arrays, one for each book. When Written content is segmented by book, the 16-bit WritIndex is appropriate for direct indexing into the segmented array of records for that book).

Lemmata content originally appeared in the 2017 Edition of the SDK. The original version obtained Lemmata from the NLTK Python library. Now Lemmata are obtained from the MorphAdorner Java service (MorphAdorner also performs all of the POS tagging). Incidentally, each Lemma ordinarily maps to multiple English words or lexemes, (e.g. 'be' is the lemma of 'are', 'were', 'is', 'art', 'wast', and 'be'). Interestingly, many words, for example 'run', are not constrained to a single uniform POS tag. Consequently, Lemmata lookup requires the POS tag. Successful lookups into Lemmata result in a list of WordKeys or OOVKeys (When a Lemma is OOV[2], it cannot be found in the Lexicon, but it can be found in the OOV-Lemmata table).

Lemmata Content (variable length records)

Part-of-Speech (POS32): uint32	Word Key uint16	PN+POS12 bits: uint16	Count uint16	Lemmata Array Uint16[] (Word or OOV keys)
0x00000036	0x0001 (a)	0x0F00	1	0x0001
0x00000094	0x0001 (a)	0x0D00	1	0x0001
0x80004206	0x0001 (a)	0x0400	1	0x0001
0x01074F9C	0x0002 (i)	0x4080	1	0x0002
…				
0x00003A1C	0x027A (elim)	0x4030	1	0x027A
0x000001DD	0x027B (elms)	0x8010	1	0x8304 (OOV: elm)
…				
0xFFFFFFFF				

OOV-Lemmata Content (lookup for OOV lemmas)

OOV Key uint16	OOV Word Length+1 bytes
0x8301	aid\0
…	
0x8F01	covenantbreaker\0

OOV (composition by example)

OOV Marker 1 bits	OOV Length 7 bits	OOV Index byte
0x8___	0x_3__	0x__01

(binary of 0x8301 is b1000001100000001)

[2] OOV stands for "Out of Vocabulary": Not all lemmas are in the AV-Lexicon; these OOV words can be looked up in the AV-Lemma-OOV table. As an example, "covenantbreakers" is in the KJV bible and therefore in the lexicon. However, covenantbreaker is not in the KJV bible (It is an example of an OOV word).

The Lexicon provides both original and modern orthographic representations for each lexeme identified in the Written content; and it includes a search representation, stripping out all hyphens. What follows are an array of associated Part-of-Speech (POS). This is an 5-bit encoded value. A reference implementation for decoding this POS value into a human readable POS string can be found in the github repo.

Lexicon Content (variable length records)

Rec # (0 bits)	Entities uint16	Size uint16	POS[0] uint32	POS[1] uint32	POS[2] uint32	• • •	POS[n-1] uint32	Search char [ ]	Display char[ ]	Modern char [ ]	
0	0xFFFF	n=2	12567	0x3112							metadata
1	0x0000	n=4	0x00000094	0x00000036	0x0000000A		0x80004206	a			Entities = { } dt, av, j, pp-f
2	0x0000	n=3	0x01074F9C	0x0000000A	0x01073F9C			i			Entities = { } pns11, j, pno11
3	0x0000	n=1	0x000002A8					o		oh	{ } oh
…											
366	0x8009	n=2	0x00003A1C	0x000740FC				adam			Entities = {Man, City} np1, npg1
…											
1311	0x0000	n=2	0x01073FBC	0x0000000A				thou		you	Entities = { } pns21, j
…											
12567	0x0000	n=1	0x0000000A					Mahershal alhashbaz	Maher-shalal-hash-baz		Entities = { } j

Entities = {Hitchcock=0x8000, men=0x1, women=0x2, tribes=0x4, cities=0x8, rivers=0x10, mountains=0x20, animals=0x40, gemstones=0x80, measurements=0x100}

NOTE: AV-Lexicon differs from Z14 release: it inserts a zeroth-record, making lex-key equal to record-index. It also differs by omitting the marker/final record after record #12567, as did the Z14 release. Otherwise, they are identical.

Additional notes about Part-of-Speech in Digital-AV

Both the PN+POS(12) field and the POS(32) field are found in Written content. And both represent Part-of-Speech, in different, but related manners. POS(12) is entirely bitwise, and therefore easier to make programmatic determinations based upon that field. POS(32) is a 5-bit encoded string. Decoding the 32-bit value into a string can be performed using the reference code cited on this page below. POS tagging was extracted from Morph-Adorner (also cited below). POS(12) is derived both from the MorphAdorner tag and innate knowledge in the Digital-AV compiler of pronouns and morphology. POS(32) is an encoded human-readable string. An earlier version of the SDK contained a HashMap, mapping each POS(32) value into a collection of POS(12) values. However, that file was deemed incomplete and has been eliminated from the SDK. That mapping might be useful, but is easily inferred from Written content. AV-Lexicon contains only POS(32) references, and no POS(12) references.

In short, the PN+POS(12) field is more granular and has a bitwise representation. Contrariwise, the encoded 32-bit POS fields have far more fidelity, but require decoding to expose their string representation.

For more information, see:

- https://github.com/kwonus/Digital-AV/blob/master/z-series/Part-of-Speech-for-Digital-AV.pdf
- https://github.com/kwonus/AVXText/blob/master/FiveBitEncoding.cs [*method signature:* string **DecodePOS**(Uint32 **encoding**)]

Names Content (variable length records)

WordKey uint16	1st Meaning	Delimiter	2nd Meaning	Delimiter	3rd Meaning	Delimiter	• • •
AVLexicon WordKey for Aaron	a teacher	|	lofty	|	mountain of…	\0	
AVLexicon WordKey for Abaddon	the destroyer	\0					
AVLexicon WordKey for Abagtha	father of the…	\0					
. . .							

AV-Names.dxi is a binary representation of "Hitchcock's Bible Names Dictionary", authored by Roswell D. Hitchcock in 1869. The difference here is that it is integrated by indexing with the word-key found in AV-Lexicon.

OVERALL PROJECT STATUS:

It's an exciting time at AV Text Ministries, and if you want to lend a hand. Let us know your technical skills and interests and we can help jumpstart you onto the team. We are embarking on brand-new support for Rust. Currently, AV Text Ministries is 100% volunteer, so if you don't just have passion about the mission as your raw motivation, it might not be the best fit.

Finally, on the non-technical side of things, we would certainly welcome a ministry sponsor that would want to place AV Text Ministries under the banner of their own local church ministry. Check http://avtext.org to discover our overall vision.

HOW THE DIGITAL-AV "PLATES" ARE AUTHORED:

Initially, various publicly available KJV texts were parsed and dutifully compared (comparing scripture with scripture [1 Corinthians 2:13]). That work produced the freeware program, AV-1995 for Windows; it was written in Delphi/Pascal and was maintained until the AV-2011 release. In 2008, the initial Digital-AV SDK was conceived and produced, harvesting much of the inner workings of AV-2008 and utilized RemObjects Oxygene/Pascal as the development platform. It was released as open source. Later, AV-2011 was "compiled" using AV-2008 as a baseline. Subsequently, the 2017/2018 Editions were "compiled" using AV-2011 as a baseline. The Z07 release of the SDK were baselined from AV-2018 edition using the K817 release. C# is now the programming language of the SDK compiler; and the ancient pascal sources were finally retired (replaced by C# sources) in 2018. The SDK-compiler uses MorpAdorner[3] (written in Java 1.6), along with the NUPOS [4]tag-set. NLTK[5] (Python) is used when MorphAdorner encounters a word out of its vocabulary. Java and Python dependencies are not exhibited in the delivered SDK (They are only part of the compilation process for the published SDK).

The Z-Series introduced a new versioning scheme, but it was mostly unchanged from the 2018 SDK Release. All new versions are compiled using the previous Z-Series release as a baseline. The new Omega-Series was compiled using Z-Series assets. For the first time, the revision to 3.5 of the Omega release utilized the 3.2 Omega rlease as its baseline. We will likely retrofit the 3.5 release into the C# Z-series pipeline. However, only the http://github/AV-Text/AVX has the Omega 3.5 release, as of May 2023. The Omega releases were inspired by the simplicity of utilizing Flat Buffers: why mess with a bunch of files when we can mess with just one?

[3] http://morphadorner.northwestern.edu/morphadorner/
[4] https://github.com/kwonus/Digital-AV/blob/master/z-series/Part-of-Speech-for-Digital-AV.pdf
[5] http://www.nltk.org

Digital AV SDK – Record layouts & File inventory

Release Version: Omega-3.5

VERSION IDENTIFIERS

Digital-AV SDK: Ω35$_7$

SDK Document: Ω39$_1$

LICENSE REQUIREMENT:

- In order to comply with the MIT-style open-source license, please include AV-License.txt with your distribution of any file identified in this SDK. The text of that file, as of 2023, is provided also at the bottom of this page.

All SDK artifacts are on github.com:

https://github.com/AV-Text/AVX

IMPROVEMENTS & CAVEATS:

- Underlying SDK formats have stabilized in the Z-series and Ω-series editions.
- The huge difference between the Z-series and Ω-series editions is that Omega edition utilizes a single file for deserialization.
- Ω32 release introduces revised Written, Book, and Chapter content
- Ω32 eliminates discrete content. Instead, that is provided directly in the Written content.
- Ω-series editions replaces the AV-Inventory file with Directory content, which is the first data payload of the new deserialization file.
- The name of the serialization file is **AVX-Omega.data**; and **AVX-Omega.md5** contains a hash of the data file.
- For the most part, the Omega releases share most of the same formats as the earlier Z-Series SDK. Ω35+ is the recommended SDK for future development.
- Hashing values in Directory & total-verse-counts for each Book were incorrect in Ω32. This necessitated the Ω35 release. Only Directory & Book content was revised.

ADDITIONAL RELEASE NOTES:

#1 Digital-AV revision numbers use a three-digit character sequence, plus an optional suffix/subscript. Revision numbers begin with the letter **Z** or **Ω**. The next two characters represent year and month of the revision. The character sequence is **X*ym*** where X is either **Z** or **Ω**, indicating either "Z-series" or "Ω-series" of the SDK (this also distinguishes the release from older Digital-AV SDK editions); ***y*** represents the year, and ***m*** represents the month. ***y*** encodes the year as a single base-36 digit; For example, (*y*=0) represents 2020; (*y* == 3) represents 2023; (*y* == 5) represents 2025; (*y* == A) represents 2030; (*y* == F) represents 2035; (y == Z) represents 2055. With respect to months, digits 1 through 9 are as expected; (*m* == A) is October; (*m* == B) is November; and (*m* == C) is December. An optional single letter/number subscript is usually included. If the subscript is a Greek letter (α or β), then this is alpha or beta. Subscript *x* indicates that it is soon to be defunct. Otherwise, subscript is calendar day of the release, encoded in base-32; the 1st→*1*, 2nd→*2*, … , 9th→*9*, 10th→a, 11th→b, 12th→c, … , 31st→*v*.

#2 Multiple revision numbers exist: The Digital-AV SDK revision (aka, the "plate" revision) is the most significant set of files. There are also distinct and separate revision numbers of this document itself. Finally, when an appendix is includes, those also have distinct revision numbers.

#3 Foundational support for Rust and C++ is provided with static initializers for all SDK content. These are experimental: while they compile, they produce stack overflow errors at runtime. Still, they are useful text representations of what the SDK looks like from an API standpoint. Foundations/csharp/avxlib actually provides a working implementation. It does not use initializers as the Rust and C++ experimental implementations do. It deserializes the content in accordance with this spec. There are some hard-coded values in the library. It should be treated as an alpha release, but it does work.

Additional information available at: http://Digital-AV.org , http://AVText.org , info@avtext.org , kevin@wonus.com

Digital AV SDK – Rust Foundational Support

Version: 3.2

No deserialization required! That's right, the Rust sources have the entire SDK files baked into the source code with requisite native array initializations. Just include the dependency in cargo, and you're good to go.

Rust sources can be found in the Digital-AV/z-series/foundations/rust/ folder on GitHub. All structures are pre-defined in lockstep with the binary files of the SDK. However, one major deviation is that the AV-Writ.dx file is segmented into 66 different structures (one for each book of the bible).

There are other minor deviations from the baseline SDK documentation. These are driven somewhat by the syntax of Rust, and to simplify code-generation. Deviations should be intuitive by comparing the struct definitions with the SDK documentation. The value of the generated files is that no deserialization operations are needed. Again, the implementation uses Rust arrays with static initializers.

The code currently compiles, but is largely untested.

The compiled Rust library is almost 400mb. That's twenty times the size of the baseline [serialized] SDK files. At first glance, this might lead you to the C++ library. However, this would be an apples to oranges comparison. The C++ implementation is a DLL (i.e. a shared library). The Rust library is static, by convention, with all dependencies baked in, including the Rust runtime itself. Someone could measure what the library would be if it were compiled as a shared library, but I have no plans to do that. For what it is, and given modern hardware, 400 mb is not very large by database standards. Yet, if trimming down is your goal, not every file need be included in your application.

Digital AV SDK – C++ Foundational Support

Version: 3.2

VERSION IDENTIFIER
C++ Support: $\Omega 32_{\alpha}$

No deserialization required! That's right, the C++ sources have the entire SDK files baked into the source code with requisite native C++ array initializations. Just include the dependency in CMake, and you're good to go.

C++ sources can be found in the Digital-AV/z-series/foundations/cpp/ folder on GitHub. All structures are pre-defined in lockstep with the binary files of the SDK. However, one major deviation is that the AV-Writ.dx file is segmented into 66 different structures (one for each book of the bible).

There are other minor deviations that should be intuitive by examining the struct definitions. These are driven somewhat by the syntax of C++, and to simplify code-generation. The value of the generated files is that no deserialization operations are needed. Again, the implementation uses C++ arrays with static initializations.

The code currently compiles, but is largely untested.

Interestingly, using the latest Microsoft x64 C++ compiler to compile the entire SDK into a DLL with static C++ arrays, the entire DLL weighs in at 21.2 mb, about the same as the experimental FlatBuffers content data. Compared to the Baseline SDK files themselves, that's only 2 mb of overhead (and all of the deserialization work is already done).

Digital AV SDK – C# Foundational Support
Version: 3.2

VERSION IDENTIFIER
C# Support: $\Omega 35_{\alpha}$

COMING SOON!!!

Foundational support for C# differs from the Rust and C++ implementations, as the C# foundation uses the Ω-series SDK for actual deserializations in lieu of code-generation. With the Ω-series SDK, we open just a single binary file to extract all SDK content.

C# sources can be found in the Digital-AV/omega/foundations/csharp/ folder on GitHub. As with the other foundational support, Written content is segmented into 66 different arrays and placed in the Book index/content (one slice for each book of the bible).

The code is currently in development, but leverages the rock solid foundation of decades of earlier deployments of the Digital-AV SDK.

The fundamental difference here with the companion project AVXText, is that the current AVXText github bundles the interpreter with the bible content for search capabilities. This is no longer necessary as Quelle can serve that purpose, while these sources are dedicated only for deserialization, validation, and content delivery. AVXText will eventually be replaced with an integrations of C# Foundation with Quelle which leverages a Parsing Expression Grammar (PEG) library called Pinshot-Blue and a parse interpreting service called Blueprint-Blue.