Digital AV SDK – Record layouts & File inventory

Revision: 3.2

REVISION IDENTIFIER

Digital-AV SDK: $\Omega 32_c$

SDK Document: $\Omega 32_c$

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 foundations 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 uint128
Directory	0	432	48	9	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Revision	432	4	4	1	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Book	436	3,216	48	67	2FA48823BF12F70E7793E0CA4D451A35
Chapter	3,652	95,120	8	11,890	BDE15818272ACDB3EE16DF7F9D136F53
Written	98,772	18,951,624	24	789,651	49F2B81AC6BDA0C7B5BCCE1D4EFBD686
Lexicon	19,050,396	246,258	0	12,567	F1C7694D3C5B15A526845D7A4946BDFE
Lemmata	19,296,654	182,344	0	15,171	B64F907ABC54470F2D227D8AC5703E33
OOV-Lemmata	19,478,998	7,754	0	771	ADEA45027082EC56EA59B079EF94C96F
Names	19,486,752	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 release are derived from the earlier Z-Series releases. 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 more than a half dozen files). In the table above, record length of zero (0), indicates that the record is variable length. A non-zero length means that the record is fixed length. Each content section has an offset and length (in bytes). Record count and content hash using MD5 are also included in the directory.

The file format defined in this document pertains to the Ω-Series releases. Both Ω and Z series formats are substantially identical (the main difference is the number of files).

Digital AV SDK – Record layouts & File inventory

Revision: 3.2

REVISION IDENTIFIER

Digital-AV SDK: $\Omega 32_c$

SDK Document: $\Omega 32_c$

Revision Content (4 bytes)

Revision *uint32*
0x3203

The revision record encodes the date of the release in four bytes.

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	0	0	0	0x3203	0	0	0	0xC0C94	24
1	0x391C 0 0 0	1:1:1:10	0x8___	0x0015 (in)	0x00	0xE8	0x00E0	0x40080470	0x0015
2	0x391C 0 0 0	1:1:1:9	0x0___	0x0136 (the)	0x00	0x00	0x0D00	0x00000094	0x0136
3	0x391C 0 0 0	1:1:1:8	0x0___	0x24F9 (beginning)	0x00	0x00	0x4010	0x000001DC	0x24F9
…	<< Beginning of Genesis 1 depicted above >>								
0xBDDBA	0x25A0 0 0 0	66:22:21:35	0x8___	0x0136 (the)	0x00	0xE0	0x0D00	0x00000094	0x0136
0xBDDBB	0x25A0 0 0 0	66:22:21:34	0x8___	0x2CB2 (revelation)	0x00	0x00	0x4010	0x000001DC	0x2CB2
0xBDDBC	0x0978 0 0 0	66:22:21:33	0x0___	0x001D (of)	0x00	0x00	0x0400	0x80004206	0x001D
…	<< Beginning of Revelation 1 depicted above >>								
0xC0C92	0x1460 0 0 0	66:22:21:3	0x0___	0x015C (you)	0x00	0x00	0x20C0	0x00083BBD	0x015C
0xC0C93	0x0F74 0 0 0	66:22:21:2	0x0___	0x0036 (all)	0xE0	0x04	0x0D00	0x00000004	0x0036
0xC0C94	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 found in the direct 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 #

The Q32 release eliminates the AV-Verse index and places 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 with 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 two 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***.

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

[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.

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.

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 Segmer	. ? !
Core Segmen	:
Real Segmen	. ? ! :
Soft Segment	, ; ( ) --

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, verse-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*	Verse Count *uint16*	Writ Count *uint32*	Writ Index *uint32*	Book Name *16 bytes* (utf8)	Abbreviations (utf8) a2 a3 a4 *(9 bytes)*	Abbreviations (utf8) Alternates *(9 bytes)*
0	0	0	0	48	0x3203	Z32c----------	-- --- ----	Revision-
1	50	0	1533	38262	0	Genesis---------	Ge Gen Gen-	Gn-------
2	40	50	1213	32685	38262	Exodus----------	Ex Exo Exod	---------
3	27	90	859	24541	70947	Leviticus-------	Le Lev Lev-	Lv-------
66	22	1167	404	11995	777656	Revelation------	Re 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 (8 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, superceding the Verse-Index found in Z31. 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	0x3203							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.Z32 differs from Z14 revision: 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 Revision. 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 AV-Writ.dx. 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 from AV-Writ.dx. 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. In 2008, the initial Digital-AV SDK was conceived and produced, harvesting much of the inner workings of AV-2008, utilizing RemObjects Oxygene/Pascal as a development platform and releasing it 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 revision of the SDK were baselined from AV-2018 edition using the K817 revision. 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) and the NUPOS [4]tag-set. NLTK[5] (Python) 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).

The Z-Series introduced a new versioning scheme, but were substantially similar to the 2018 SDK Release. All new versions are compiled using the previous Z-Series release as a baseline. The new Omega-Series are compiled using Z31. However, effectively, the work to compile the initial two Z-series releases were the recent heavy lift. All releases since Z14 have been reformatting exercises. The Omega releases were inspired by the simplicity of utilizing the Flat Buffers release: 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

Revision: 3.2

REVISION IDENTIFIER

Digital-AV SDK: $\Omega 32_c$

SDK Document: $\Omega 32_c$

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/kwonus/Digital-AV

IMPROVEMENTS & CAVEATS:

- Fundamental SDK format has stabilized in the Z-series and Ω-series revisions.
- The Ω32 release introduces revised Written, Book, and Chapter content and a more intuitive directory for all content. Otherwise, the Ω32 release inherits all of the earlier Z-Series enhancements.

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 1^{st}→*1*, 2^{nd}→*2*, … , 9^{th}→*9*, 10^{th}→a, 11^{th}→b, 12^{th}→c, … , 31^{st}→*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 Not all files in this SDK are required to produce working bible software. Some of the information in the index files is redundant, only reducing lookup complexity. In fact, with just the Written, Book, and Lexicon content, there is enough information to print the whole bible, including chapter and verse numbers. However, the addition of Chapter content can simplify processing. Additional SDK content serves as reference material for the baseline content.

#5 Foundational support for Rust and C++ is now provided. Appendices, which follow, provide overall status. *FoundationsGenerator.csproj* in the Z-Series/generator folder (within the GitHub repo), is how the Rust and C++ source code is generated. Flat Buffers and Protocol Buffers are in early development also

Digital AV SDK – Rust Foundational Support

Revision: 3.1

REVISION IDENTIFIER

Digital-AV SDK: $Z32_{\alpha}$

Rust Support: $Z32_{c}$

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

Revision: 3.1

REVISION IDENTIFIER

Digital-AV SDK: $Z32_{\alpha}$

C++ Support: $Z32_{c}$

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 3 mb of overhead (and all of the deserialization work is already done).

Digital AV SDK – Experimental Protocol Buffer Support

Revision: 3.x

REVISION IDENTIFIER

Digital-AV SDK: $Z31_{\alpha}$

ProtoBuf Support: $Z31_{q}$

This is the first release with Protocol Buffer (protobuf) support. Some quirks are manifested in supporting protobuf because the serialization format has no support for uint16 or byte fields. Even the on-disk format is porky. The bloat would be excessive in highly-repeated messages after deserialization into RAM without some mitigation. Therefore, a few of the highly-repeated message types conflate adjacent fields into uint32. In smaller tables, this is not done. It is recommended to have getters on the deserialized classes that fetch discrete elements of these conflated fields. Where this has occurred is obvious in the IDL (details can be found in the ProtoGen.csproj itself (in ProtoGen.cs). Serialized data is more than twice the size of Flat Buffers. Deserialized data will have even more bloat.

Baseline AV SDK item	Baseline Size	ProtoBuf IDL	ProtoBuf binary content	ProtoBuf Size
AV-Writ.dx	17 mb			
AV-Book.ix	3 kb			
AV-Chapter.ix	12 kb			
AV-Verse.ix	122 kb			
AV-Lemma.dxi	179 kb			
AV-Lemma-OOV.dxi	8 kb			
AV-Lexicon.dxi	241 kb			
AV-Names.dxi	60 kb			
Total	18 mb	avx.proto	avx-protobuf.data	44 mb

Digital AV SDK – Experimental Flat Buffers Support

Revision: 3.1

REVISION IDENTIFIER
Digital-AV SDK: $Z31_{\alpha}$
Flat Buffers Support: $Z31_{q}$

If the developer is willing to take on the dependency of Flat Buffers[6], the deserialization can be driven using a single IDL file named avx.fbs. The content file is named avx-fb.data. The layouts are substantially similar to the baseline SDK. Therefore, the baseline SDK documentation can still be consulted. However, deserialization is driven through Flat Buffers, and is compatible with most programming languages.

The files in the table below are consistent with the latest revision of the baseline SDK. The fundamental difference is the serialization format itself.

Baseline AV SDK item	Baseline Size	Flatbuffer IDL	FlatBuffer binary content	FlatBuffer Size
AV-Writ.dx	17 mb			
AV-Book.ix	3 kb			
AV-Chapter.ix	12 kb			
AV-Verse.ix	122 kb			
AV-Lemma.dxi	179 kb			
AV-Lemma-OOV.dxi	8 kb			
AV-Lexicon.dxi	241 kb			
AV-Names.dxi	60 kb			
Total	18 mb	avx.fbs	avx-fb.data	21 mb

FlatBuffers-special files can be found in the FB sub-folder of the Z-Series SDK[7]. These two files have been written using FlatSharp[8]. As of the date of this documentation, Flat Buffers assets should be considered Alpha-quality. They are available for use, but completely untested as yet.

The status of FB support is experimental. Course metrics show the Flat Buffers content, weighing in at 21mb, induces less than 2% size overhead vis-à-vis the baseline SDK. The convenience here, at least with FlatSharp, is just a few lines of code and a single file to deserialize.

[6] See https://google.github.io/flatbuffers/
[7] See https://github.com/kwonus/Digital-AV/tree/master/z-series/FB
[8] See https://github.com/jamescourtney/FlatSharp