Digital AV SDK – Record layouts & File inventory

Revision: 3.2

REVISION IDENTIFIER
Digital-AV SDK: $Z32_n$
SDK Document: $Z32_n$

Digital-AV Software Development Kit provides the foundation for a fully working bible application, with no external dependencies. In fact, implementation with fewer than 1000 lines of code is possible, as demonstrated by the golang sources in this SDK. The SDK provides everything, including data and index files. Some developers have discovered that they can be up and running in under an hour. Easily jumpstart your development by working with the provided golang sources, or go all in from scratch with the programming language of your choice.

BREAKING NEWS: Direct support for Rust, C++, and Flat Buffers debuted in the Z-3.x revisions. See the addendums at the bottom of this document for status of each of these endeavors.

The base Digital-AV SDK (AVSDK) is entirely file based. There are zero dependencies and zero language bias (all programming languages can read files). File formats defined in this document use a consistent naming convention: the extent of each data file reveals the content and record type. The table to the right, defines the various extents of files that compose the SDK.

File Extent	File Type	Record Type	Content Type
*.dx	data	fixed length	binary
*.ix	index	fixed length	binary
*.dxi	data + index	variable length	binary
*.bom	MD5 checksums	newline delimited	text
*.md5	MD5 checksum	single text field	text
*.ascii	informational	newline delimited	text

File extents identify the format/file type. Fixed-width data files have a *.dx file extent, while fixed-width index files have an *.ix file extent. A third file-type combines the data and index into a single file (and those binary files are always variable length). The *.bom file contains MD5 hashes which can be used at runtime to verify the file conforms to the release. The *.ascii files provide information for the developer and are not expected to be deployed to the end-user.

AV-Writ-*.dx files are three variants that handle disparate memory and/or system constraints. While the file formats are detailed later in this document, the table to the right is provided to summarize the records widths of each variant. (you need only deploy a single AV-Writ-* file with your application). It is up to the developer to weigh the footprint versus features in that decision

AV-Writ variants	Size in bytes
AV-Writ-22.dx	22 bytes
AV-Writ-16.dx	16 bytes
AV-Writ-04.dx	4 bytes

Digital AV SDK – Record layouts & File inventory

Revision: 3.2

REVISION IDENTIFIER
Digital-AV SDK: $Z32_n$
SDK Document: $Z32_n$

The weightiest data files are those named AV-Writ-*.dx; these files contain a stream of words for each verse of each chapter of each book. These are not text files. Therefore, they are quite compact. Several fields are index lookups into other SDK files. The entire inventory of files implements an efficient database of word embeddings that are compactly manifested in RAM. Of course, The AV-Writ-*.dx files with the widest records are obviously also the most feature rich.

AV-Writ-22.dx (22 bytes per record)

Record # *0 bits*	Hebrew | Greek *4 x uint16*	Verse *uint16*	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	0	0x8___	0x0015 (in)	0x00	0xE8	0x00E0	0x40080470	0x0015
1	0x391C 0 0 0	0	0x0___	0x0136 (the)	0x00	0x00	0x0D00	0x00000094	0x0136
2	0x391C 0 0 0	0	0x0___	0x24F9 (beginning)	0x00	0x00	0x4010	0x000001DC	0x24F9
…	<< Beginning of Genesis 1 depicted above >>								
0xBDDB9	0x25A0 0 0 0	30698	0x8___	0x0136 (the)	0x00	0xE0	0x0D00	0x00000094	0x0136
0xBDDBA	0x25A0 0 0 0	30698	0x8___	0x2CB2 (revelation)	0x00	0x00	0x4010	0x000001DC	0x2CB2
0xBDDBB	0x0978 0 0 0	30698	0x0___	0x001D (of)	0x00	0x00	0x0400	0x80004206	0x001D
…	<< Beginning of Revelation 1 depicted above >>								
0xC0C91	0x1460 0 0 0	31101	0x0___	0x015C (you)	0x00	0x00	0x20C0	0x00083BBD	0x015C
0xC0C92	0x0F74 0 0 0	31101	0x0___	0x0036 (all)	0xE0	0x04	0x0D00	0x00000004	0x0036
0xC0C93	0x0119 0 0 0	31101	0x8___	0x018A (amen)	0xE0	0xFC	0x8000	0x8000550E	0x018A
<< End of Revelation 22:21 depicted above >>									

AV-Writ-22.dx begins with ***Greek & Hebrew*** Strong's numbers in the Old & New Testament. Each English word can have up to four Strong's numbers associated with it. Strong's numbers are an integer representation of the original Hebrew/Greek words from which the English words were originally translated.

While words in the Old Testament can have a maximum of four Strong's numbers associated with a single English word.

The Omega-series SDK has a differing format for Written content. Many of the files between the Z-series SDK and the Omega-series are identical byte-streams, but this one differs.

Hebrew | Greek encodings

This is characteristic of the KJV translation, but four slots are reserved even for the Greek to maintain a fixed record width across entire bible. For more information on Strongs numbers, refer to the Strong's Exhaustive Concordance for additional backround information. Also note that ***Verse*** is an inline index-pointer to the corresponding AV-Verse index.

Strongs #1	Strongs #2	Strongs #3	Strongs #4
1st Strongs #	2nd Strongs #	3rd Strongs #	4th Strongs #

Capitalization bits and WordKey

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 of those is ***Caps***: these 2-bits 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 ***WordKey*** (a lookup key for AV-Lexicon). Incidentally, the lookup key is equally compatible with the earlier AV-Lexicons, even the AV-Lexicon-i728.dxi, from five-year-old 2018 SDK.

The next field is the ***Punctuation*** byte. Each word can have certain punctuation applied either as a prefix to the word, or alternatively as a suffix. An example of prefix punctuation is an open parenthesis. There are numerous examples of suffix punctuation, such as period, comma, or close parenthesis. The puncuation byte also has bits to represent italisized words in the text and even mark the words spoken by Jesus, which some bibles represent as red-colored text.

Punctuation Byte

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

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--)

Within AV-Writ-22.dx and AV-Writ-16.dx, 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. AV-SDK encodes the markings for both person and number using the binary representation depicted in the table to the left. Similarly, the remaining twelve bits provide course part-of-speech markers.

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

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. In this edition of the SDK, these bondaries are interpretted based upon a combination of 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

* ***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.

There are two additional trimmed down versions of the AV-Writ files which contain subsets of AV-Writ-22.dx. These can be used for more memory constrained implementations or utilized where the additional data fields are not needed.

AV-Writ-16.dx (16 bytes per record)

Record 0 bits	Hebrew | Greek 4 x uint16	Verse uint16	Caps 2 bits	WordKey 14 bits	Punc byte	Transition byte	PN+POS(12) uint16
0	0x391C 0x0 0x0 0x0	0x0000	0x8___	0x0015 (in)	0x00	0xEF	0x00E0
1	0x391C 0x0 0x0 0x0	0x0000	0x0___	0x0136 (the)	0x00	0x00	0x0D00
2	0x391C 0x0 0x0 0x0	0x0000	0x0___	0x24F9 (beginning)	0x00	0x00	0x4010
…	<< Beginning of Genesis 1 depicted above >>						
C0C91	0x1460 0x0 0x0 0x0	0x797D	0x0___	0x015C (you)	0x00	0x00	0x20C0
C0C92	0x0F74 0x0 0x0 0x0	0x797D	0x0___	0x0036 (all)	0xE0	0x06	0x0D00
C0C93	0x0119 0x0 0x0 0x0	0x797D	0x8___	0x018A (amen)	0xE0	0xFE	0x8000
<< End of Revelation 22:21 depicted above >>							

AV-Writ-04.dx (4 bytes per record)

Record 0 bits	Caps 2 bits	WordKey 14 bits	Punc byte	Transitions byte
0	0x8___	0x0015 (in)	0x00	0xEF
1	0x0___	0x0136 (the)	0x00	0x00
2	0x0___	0x24F9 (beginning)	0x00	0x00
…	<< Beginning of Genesis 1 depicted above >>			
C0C91	0x0___	0x015C (you)	0x00	0x00
C0C92	0x0___	0x0036 (all)	0xE0	0x06
C0C93	0x8___	0x018A (amen)	0xE0	0xFE
<< End of Revelation 22:21 depicted above >>				

Segment Transitions

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

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

AV-Book provides indicies into AV-Chapter, AV-Verse, and AV-Writ, and corresponding chapter-counts, verse-counts, and word-counts (for each of the sixty-six books of the bible). It reserves a fixed sixteen bytes for the book-name, a fixed nine bytes (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.

AV-Book-50.ix (50 bytes)

Book Number *byte*	Chapter Count *byte*	Chapter Index *uint16*	Verse Count *uint16*	Verse Index *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	0	0	0x3112	Z31g----------	-- --- ----	Revision-
1	50	0	1533	0	38262	0	Genesis---------	Ge Gen Gen-	Gn-------
2	40	50	1213	1533	32685	38262	Exodus----------	Ex Exo Exod	---------
3	27	90	859	2746	24541	70947	Leviticus-------	Le Lev Lev-	Lv-------
66	22	1167	404	30698	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 Z-3.x releases. 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. The previous AV-Book.ix has been renamed AV-Book-Z14.ix. Either file is fully compatible with all Z-series revisions. Weighing in with only 1,238 additional bytes, the newer file likely results in fewer ancillary lookups. Note also that the Z-Series records differ with Omega on the size for WritCount.

AV-Book-32.ix (32 bytes / Z14) *Consistent with earlier SDK releases, including Z14*

Record # 0 bits	Book Number byte	Chapter Count byte	Chapter Index Uint16	Book Name 16 bytes (utf8)	Book Abbreviations 12 bytes (utf8)
0	1	50	0	Genesis---------	Ge----------
1	2	40	50	Exodus----------	Ex----------
2	3	27	90	Leviticus-------	Le----------
…					
65	66	22	1167	Revelation	Re

AV-Chapter-10.ix (10 bytes)

Record # 0 bits	Writ Index uint32	Writ Count uint16	Verse Index uint16	Verse Count uint16
0x000 (genesis:1)	0	797	0	31
0x001 (genesis:2)	797	632	31	25
0x002 (genesis:3)	1429	695	56	24
	. . .			
0x4A2 (revelation:20)	787852	477	31039	15
0x4A3 (revelation:21)	788329	749	31054	27
0x4A4 (revelation:22)	789078	573	31081	21

AV-Chapter-08.ix (8 bytes / Z14)

Record # 0 bits	Writ Index uint32	Verse Index uint16	Word Count uint16
0x000 (genesis:1)	0	0	797
0x001 (genesis:2)	797	31	632
0x002 (genesis:3)	1429	56	695
	. . .		
0x4A2 (revelation:20)	787852	31039	477
0x4A3 (revelation:21)	788329	31054	749
0x4A4 (revelation:22)	789078	31081	573

NOTE:

AV-Chapter.ix differs from earlier revisions, as it how includes verse count and an altered column order than the earlier Z14 Revision.

AV-Chapter contains one extra data field than its Z14 counterpart, either file is fully compatible with all Z-series revisions.

AV-Verse.ix (4 bytes)

Record# 0 bytes	Book, Chapter, Verse, Words 4 bytes: BB:CC:VV:WordCnt	
0x0000	1:1:1:10	In the beginning …
0x0001	1:1:2:29	And the Earth …
0x0002	1:1:3:11	And God said …
	…	
0x797B	66:22:19:44	And if any man … are written in this book.
0x797C	66:22:20:16	He which testifieth … Even so, come, Lord Jesus.
0x797D	66:22:21:12	The grace of our Lord … be with you all. Amen

The AV-Lemma file 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 server (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 in AV-Lemma result in a list of WordKeys or OOVKeys (When a Lemma is OOV[1], it cannot be found in the Lexicon, but it can be found in the OOV table).

AV-Lemma.dxi (variable length)

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				

AV-Lemma-OOV.dxi (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)

[1] 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).

AV-Lexicon provides both original and modern orthographic representations for each lexeme identified in AV-Writ. It also includes a search-version of the lexeme that strips out all hyphens. Next, an array each Part-of-Speech (POS) associated with that lexical item is identified by an encoded value using 5-bit character encoding. A reference implementation for decoding a Uint32 value into a human readable POS string can be found in the github repo.

AV-Lexicon.dxi (data and index combined: 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.Z31 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-22.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 reconstructed from AV-Writ-22.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)**]

AV-Names.dxi (data and index combined: 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.

avx.go (golang source code)

avx.go implements a web-server (HTTP server) that provides the entire text of the AV bible utilizing AVX extensions, but still uses simple semantics. Version numbers for source code are respective of the SDK Document revision numbers. The first release of avx.go, which had been updated to the z-series SDK, was the Z081 golang source-code revision.

There are a couple of URLs used for testing and validation.
They also illustrate how avx.go can be extended:

- http://localhost:2121/
- http://localhost:2121/help
- http://localhost:2121/validate

Example of GoLang source in operation may be available at avbible.net:

https://avbible.net/avx/
(the web-site above also utilizes NGINX as a reverse-proxy for HTTPS)

NOTES:

1. As the web-server is not hardened, it should be placed behind a reverse-proxy if exposed to the open Internet. This is a common pattern; Apache httpd, NGINX, Caddy, or IIS can easily be configured to serve as a reverse-proxy.
2. URL form #3 and #5 are discussed under the description of the *.avspec format
3. avx.go currently utilizes the Z14 revision of the SDK, which can be found in the github history for Digital-AV.pdf. That file will be from April 2021.

The / endpoint simply reports the release number of the optional avx.go web-server component. The /help endpoint provides primitive information about the web-service. /help can be easily replaced by developer. The /validate endpoint reports on the validity of data files in accordance with the bom (The “bom”, or bill of materials, is described in the section labelled AV-Inventory.bom later in this document). In addition to the administrative URL’s described above, here is a list of the foundational endpoints that provide the core functionality of avx.go:

1. http://localhost:2121/avx/genesis
2. http://localhost:2121/avx/genesis/1
3. http://localhost:2121/avx/gen/1?sessionID
4. http://localhost:2121/avx/rev/22?sessionID=day&amen
5. http://localhost:2121/avx/rev/22?sessionID=$FFFFFFFFFFFF
6. http://localhost:2121/avx/css/sessionID.css

All of these endpoints can be summarized as one of two types: getting the chapter of a book, or getting a CSS stylesheet. When no chapter is provided, chapter 1 is always implied. When no session identifier is provided, the resulting chapter request is decorated with the baseline stylesheet, named /css/AV-Stylesheet.css. When a session identifier is provided, the session number dictates the name of the CSS file that will decorate the chapter request. Moreover, avx.go can compile information into a CSS stylesheet. When a request is made for Genesis using the URL depicted in #3 above, a stylesheet becomes linked in the response to a stylesheet with the URL depicted in #8 above. A web-browser will make an immediate subsequent request to get the stylesheet. If /css/sessionID.css does not exists, avx.go will automatically compile a file named /css/sessionID.avspec. Similarly, but easier to understand in #4 above, the URL would generate CSS which would highlight the words **day** and **amen**. In order to maintain optimal performance, session identifiers are non-volatile. In order to overwrite a *.css files and/or *.avspec files, they must be manually deleted beforehand. Avx.go uses Z08 edition.

*.avspec file format

WordKey Count Uint16	Array of Uint16	
0xnnnn	0xnnnn Count of WordKeys is followed by WordKey list [corresponds to AV-Lexicon]	
BookChapter Uint16	**Verse Count byte** *(matching verses)*	**Array of byte**
0xbbcc	0xkk	0xkk Count of matching verses is followed by an array of Verse numbers
…		
0xbbcc	0xjj	0xjj Count of matching verses is followed by an array of Verse numbers
0x0000		

avx.go software ignores everything after the first record above. Only that first record defines the CSS file. And that first line is expanded word-for-word into highlights for each supplied wordkey. A slight variation here is that Strong's numbers will eventually also support highlighting. To highlight Strong's numbers, set the 0x8000 bit for Hebrew and the 0x4000 bit for Greek. The URL form that was depicted with this syntax, sessionID=$FFFFFFFFFFFF, is primarily intended for testing. Here, the hex digits that follow the dollar sign ($) are expected to be expansions of the format described above (No record separators, just a representation of the raw bytes described above, in Big-Endian order).

AV-Stylesheet.css (text file containing CSS for avx.go; optional)

This standard-format CSS stylesheet should be included when avx.go is utilized in your development. This optional stylesheet is included in the SDK, but it can be customized in any way by the web designer. However, the web designer should realize that any references in the CSS to image files will result in 404 errors unless support is explicitly added to avx.go by your development team. Finally, avx.go always links chapter output to the AV-Stylesheet.css stylesheet, even when a *.avspec derived stylesheet is also specified.

AV-Inventory-Z31.bom (text file which identifies core inventory)

This is an ascii text file that provides a bill-of-materials for the delivered files. For each file of the SDK, the bom contains a line item for the artifact. Each line has five fields, separated by whitespace:

1. Name of the file
2. MD5 hash (hexadecimal representation) of the file
3. Record Length (decimal representation // uint32) of the file // 0 for variable-width files
4. Record Count (decimal representation // uint32) of the file
5. Size in bytes (decimal representation // uint32) of the file

The avx.go server implements a validation function, using an older bom; it reads the bom and reports inconsistencies. To avoid malicious attack, utilization of the bom is highly recommended, but not required. It helps mitigate corruption, both intentional and unintentional.

AV-Inventory-Z31.md5 (new with the Z31 release)

This ascii text file contains the MD5 of AV-Inventory-Z31.bom. For utmost security, utilized this MD5 to check the validity of AV-Inventory-Z31.bom, in addition to checking the validity of each file utilized at runtime from the bom.

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[2] (written in Java 1.6) and the NUPOS [3]tag-set. NLTK[4] (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).

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

Digital AV SDK – Record layouts & File inventory

Revision: 3.2

REVISION IDENTIFIER

Digital-AV SDK: $Z32_n$

SDK Document: $Z32_n$

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 revisions.
- The Z14 release introduced a revised AV-Writ.dx file. The Z32 release has revised it again. This one has been renamed to AV-Writ-22. Likewise, Z31 release has renamed AV-Book.ix to AV-Book-50.ix; and AV-Chapter.ix to AV-Chapter-50.ix. . There are tweaks to AV-Lexicon.dxi, and all renamed files.
- With all z-series revisions, MorphAdorner is used for part-of-speech tagging instead of NLTK. NLTK is only referenced when/if MorpAdorner fails to generate a tag.

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 ***Xym*** 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, there are distinct revision numbers of each appendix within this document.

#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 complexity. In fact, with just the AV-Book.ix, AV-Lexicon.dxi, and any one of the AV-Writ-*.dx files would be enough information to print the whole bible, including chapter and verse numbers. However, the addition of AV-Chapter.ix and AV-Verse.ix can greatly simplify processing. Additional SDK files are completely optional and serve as lookups for lemmata, person-names, and Part-of-Speech metadata.

#4 Some of the binary files have corresponding text files with *.ascii extent. Newer SDK files no longer generate these, as the C++ or Rust sources, effectively shed the same light on the contents of the newer binaries.

#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: $Z31_{\alpha}$

Rust Support: $Z31_{q}$

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-22.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: $Z31_{\alpha}$

C++ Support: $Z31_{q}$

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-22.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-22.dx	17 mb			
AV-Book-50.ix	3 kb			
AV-Chapter-10.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[5], 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-22.dx	17 mb			
AV-Book-50.ix	3 kb			
AV-Chapter-10.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[6]. These two files have been written using FlatSharp[7]. 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.

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