**STEP module converter**

Module converter



**User and maintenance Guide**

**Contents**

[1 Introduction 4](#_Toc160005310)

[1.1 The input formats 4](#_Toc160005311)

[1.2 External-facing OSIS 5](#_Toc160005312)

[1.3 Internal-facing OSIS 6](#_Toc160005313)

[1.4 Cross-boundary markup 6](#_Toc160005314)

[1.5 Reversification 7](#_Toc160005315)

[1.6 Osis2mod 8](#_Toc160005316)

[2 Using the converter – process overview 9](#_Toc160005317)

[3 Installing the tool and configuring the environment 10](#_Toc160005318)

[4 Creating a folder structure 11](#_Toc160005319)

[4.1 The per-text root folder 11](#_Toc160005320)

[4.2 The content of the per-text root folder 11](#_Toc160005321)

[5 Configuration and metadata 13](#_Toc160005322)

[5.1 Overview 13](#_Toc160005323)

[6 Do you need to pre-process the text? 14](#_Toc160005324)

[6.1 VerseLine 14](#_Toc160005325)

[6.2 USX 14](#_Toc160005326)

[6.3 OSIS 15](#_Toc160005327)

[7 Running the converter from the command line 16](#_Toc160005328)

[7.1 Generating a Sword module 16](#_Toc160005329)

[7.2 Evaluating versification schemes 18](#_Toc160005330)

[7.3 Checking the inputs for a module 18](#_Toc160005331)

[7.4 Version numbering and history information 19](#_Toc160005332)

[7.5 Module naming / reversification / use of STEP’s own version of *osis2mod* 19](#_Toc160005333)

[7.6 Checking the outputs 21](#_Toc160005334)

[7.7 The Sword module 21](#_Toc160005335)

[7.8 Additional information: the TextFeatures folder and the enhanced Sword configuration file 21](#_Toc160005336)

[8 Philosophy and implementation 23](#_Toc160005337)

[8.1 Overview 23](#_Toc160005338)

[8.2 Code structure – background 23](#_Toc160005339)

[8.3 Code structure 24](#_Toc160005340)

[8.3.1 Main 24](#_Toc160005341)

[8.3.2 ProcessingElements 24](#_Toc160005342)

[8.3.3 Subelements 24](#_Toc160005343)

[8.3.4 support 25](#_Toc160005344)

[8.3.5 osisinputonly / usxinputonly 25](#_Toc160005345)

[8.3.6 utils 25](#_Toc160005346)

[9 Tools etc 26](#_Toc160005347)

[10 OSIS conversion 27](#_Toc160005348)

[10.1 Non-compliance 27](#_Toc160005349)

[10.2 Conversion philosophy 27](#_Toc160005350)

[10.3 STEP OSIS 27](#_Toc160005351)

[10.4 A special note on tables 29](#_Toc160005352)

[10.5 A special note on canonical headings 29](#_Toc160005353)

[10.6 A special note on cross-references 30](#_Toc160005354)

[11 A note on debugging 32](#_Toc160005355)

[12 Gotchas and arcane information 33](#_Toc160005356)

[13 Intellij IDEA 34](#_Toc160005357)

[13.1 Run configurations 34](#_Toc160005358)

[13.2 Artifacts 34](#_Toc160005359)

[13.3 Bugs 35](#_Toc160005360)

# Introduction

This document talks about the STEP Text Converter. Here’s a useful schematic showing what it does, which will help you distinguish it from other lesser forms of software:

|  |
| --- |
| Module or repository package  VL  USX  OSIS  **Magic** |

In other words, it takes Bible texts in USX, VerseLine (VL) or OSIS form, and converts them to Sword modules. These formats are discussed briefly in section 1.1.

In theory any of the inputs can be either a single file or a collection of files. In practice, to date all VL and OSIS inputs have involved just a single file; and all USX inputs have comprised of a separate file per Bible book. Chances are, therefore, that these are the only arrangements which will have been tested. Don’t say you haven’t been warned.

|  |
| --- |
| **If your main aim is merely to *use* the converter, feel free to skip to section 2. If you need to understand or maintain it, read on.** |

Returning to the schematic above, can it really be that simple you ask? To which the answer is, of course no. There wouldn’t be a 40-page user and maintenance guide and 10,000+ lines of code if it could.

What, then is going on in the magic section?

First off, all of the inputs are converted to a common form which I will refer to here as ***external-facing OSIS***. External-facing OSIS is OSIS which we could make available to a third party should we wish to do so, and should licence conditions permit; and it is also OSIS which we ourselves might want to use as input in a future run. (In fact, strictly speaking we *can’t* make it available to third parties – see section 1.2.)

A copy of this external-facing OSIS is then modified in various ways to create ***internal-facing OSIS***. This is the form from which the Sword module is generated. See section 1.3 for more details.

## The input formats

**USX** is an XML dialect maintained by UBS, and still under active development. It is an XML-ified version of USFM, which is also controlled by UBS. We can indirectly accept USFM as an input as well, because the UBS Paratext tool can be used to convert USFM to USX. The various extant versions of USX differ in some significant ways, and we cannot rely upon all translators using the most recent version

Most of the texts I have seen have been in USX format. I’m not sure whether that means that most texts *are* in USX these days, or whether it just means I need to get out more.

**VerseLine** (VL) is a plain-text format in which all of the information for a single verse occupies a single line in the file. Unfortunately, that’s as far as standardisation goes, every man doing that which is right in his own eyes: no two texts seem to agree upon how to indicate the scripture reference for each verse or what other features should be supported (footnotes, etc). The converter has built-in support for the two forms of VL we have processed most recently, this support being defined using configuration data. The fact that I rely upon configuration data here holds out some hope that we might be able to support other forms of VL too with just appropriate changes to the configuration data, rather than having to augment the processing. Just not *much* hope, I suspect.

**OSIS** is another XML dialect. At the time of writing the standard is moribund (ie no one is actively maintaining it and I don’t believe any updates have been applied for a very long time). It is of interest for three reasons: first, for a few texts OSIS is all we have available; second, we may need to apply manual tweaks to some texts (eg to apply tagging), and OSIS is regarded as the best place to do this;[[1]](#footnote-1) and third, we have to generate OSIS in order to use the third party *osis2mod* tool which is our only way of making the Sword modules we need.

## External-facing OSIS

This is a form of OSIS which could possibly be made available to third parties, and which we might ourselves want to use as the basis of input to a future run (to which end the processing stores it in the *InputOsis* folder for future reference once it has been generated).

In fact, there are a couple of things which may stand in the way of making material available to third parties.

First, the OSIS is not *quite* compliant. Poetry lines and list items in OSIS are supposed to reside within enclosing tags equivalent to HTML’s <ul>. USX (which is the most common form of input at present) does not have these. It is difficult to add them reliably; doing so increases the likelihood of hitting cross-boundary markup; if we have them, they introduce excessive vertical whitespace into the rendered text; and things seem to work perfectly well without them. In view of all of this, I don’t attempt to generate them. Which, as I say, means we aren’t quite compliant (and this unfortunately also means that we cannot supply modules to Crosswire, who require full compliance).

The other issue is metadata. Of course third-party requirements in this respect are unknown anyway, but I imagine the most likely requirement would be to have a copy of relevant data from the Sword configuration file. Here we follow what is a reasonably common practice, and give certain elements in HTML form (partly because for long text like copyright details, this gives a better rendered appearance, and partly because many of our texts come from DBL, and DBL’s *metadata.xml* file often contains the data we need in HTML form, which I pick up automatically).

In theory, HTML should not be used here: we are supposed to limit ourselves to Crosswire’s own markdown. That, however, is rather restrictive.

## Internal-facing OSIS

This is the form of OSIS which is used to generate the Sword module. Its sole purpose is to make this possible: it has no long-term value, although I may retain it locally for debugging purposes.

Internal OSIS starts out as a copy of external OSIS, but is then subject to a number of modifications, which can be broadly categorised as follows:

* *Validation*: Checking for the validity of reference markings, cross-references, etc. In some cases this may result in changes to the content – for example cross-references which point to places which do not exist in the text (eg refs to the OT in an NT-only text) are converted to plain footnotes.
* *Standardisation and fixes*: Addressing things which may be wrong in the input data, or which commonly differ in different texts, or where OSIS offers more than one way of doing the same thing.
* *Added value*: For example, adding explanatory footnotes where verse numbers differ between texts.
* *Mechanics*: Things needed to help the overall process or to make *osis2mod* or STEPBible work properly. For example, restructuring material to avoid cross-boundary markup, or changing the verse structure of the text.[[2]](#footnote-2)
* *Circumventing bugs*: STEPBible appears to contain some rendering bugs, and also renders some markup in ways we do not find particularly attractive. I make changes to try to circumvent these issues. (These changes are often fairly arbitrary and, indeed, implausible: the only criterion is that they do what is required, not necessarily that they make sense.)

I keep this version of OSIS separate from externally-facing OSIS for two reasons.

First, the kinds of changes I make here are at best probably of no interest to third parties, should we wish to supply them with OSIS, and at worst probably make the OSIS of less value to them.

And second, we may wish to use OSIS as an input on future runs (for example because it is convenient to tweak it to add tagging). I have no control over the changes which that may entail, and it would become confusing if the starting point for this were internal OSIS, because I’d have to find a way to avoid applying ‘internalising’ changes to those parts of the OSIS which had already previously been internalised, whilst at the same time apply such changes to parts which had not.

## Cross-boundary markup

USX and OSIS differ somewhat as regards the manner in which books, chapters and verses are demarcated. My recollection is slightly hazy, and I can’t be bothered to look things up, but I think that all three levels (books, chapters, and verses) may be marked in USX simply with a milestone marker at the front of the given entity. OSIS, as I recall, requires that books and chapters be enclosing nodes, and allows verses to be either milestones or enclosing tags, but recommends the former.

Books and chapters can easily be swapped from one arrangement to the other because there is normally no cross-boundary markup. But the milestone form markup for verses is more problematical, because it permits, and by implication encourages, semantic and formatting markup to cross verse boundaries (ie to start outside a given verse and finish inside it, or vice versa).

This is fine for printed works, which presumably is what it is largely aimed at. But it is very awkward for electronic texts where, as with STEP, there may be a requirement to access verses individually out of context, because it becomes very difficult to excise them from their surrounding markup.

I do what I can to address this by removing any existing verse-end markers and then attempting to replace them in ‘optimal’ positions to avoid such cross-boundary markup. This is possible because there is often a fair bit of flexibility as regards their positioning. Clearly the verse-end for verse *n* must come before the verse-start for verse *n* + 1, but I am free to position the verse-end anywhere between the two verse starts, provided only that no canonical text falls outside the verse.

Unfortunately there are limits to what can be achieved like this, and I do engage in some slightly more significant surgery. In particular, I replace enclosing plain vanilla paragraphs by a self-closing paragraph marker at the front of the original paragraph, and do the same with poetry paragraphs; and where tables span multiple verses, I turn the table into a large elision and put the entire text of the table into just one of the verses making up the elision.

In fact, even if I didn’t do this, I believe that *osis2mod* does the same itself in respect of plain vanilla and poetry paragraphs (but I think not in respect of tables – I’m not 100% sure not whether it can cope with cross-verse tables). Given that osis2mod does this anyway, it rather begs the question of why I bother to do it myself. Partly this is because it means we have control of the process; and partly it is because things like reversification rely upon there being no cross-boundary markup, and reversification runs before we get as far as *osis2mod*.

## Reversification

Different Bibles label and organise verses differently. The translation of a given chunk of ancient text may be labelled Dan 1:2 in one Bible, and something different in another. This is a problem for STEP’s added value features. For instance, the vocabulary STEP displays for Dan 1:2 is based the mouse hovering over a verse marker which reads ‘Dan 1:2’. If Dan 1:2 in different Bibles reflects different underlying ancient texts, the vocabulary would – without further work – be wrong for some of them.

To address this, there are two options.

One, which I refer to as *conversion-time* reversification, entails restructuring the text during the conversion process, moving verses around and relabelling them, so that the generated module is 100% NRSV(A) compliant (NRSV(A) being our chosen standard). In general, licence conditions preclude quite such major surgery, so this is limited to relatively few texts (probably public domain ones), and in addition will probably be applied mainly to texts of interest to an academic audience, who would understand the need for reversification.

The alternative – *runtime* conversion – entails leaving the text as-is through the conversion process (which probably means it conforms neither to NRSV(A) nor to any of the other schemes built into the Crosswire *osis2mod* utility). This lack of conformity means the result has to be run through our own version of *osis2mod* which can handle texts in this form (and used with our own version of JSword which can also cope). In this case, displaying the text standalone in STEP, users are presented with the text in its native form. Divergences from NRSV(A) are taken into account only on-the-fly if the user invokes added value features which would be affected by them. This approach helps ensure we conform to licence conditions, and also produces a text which is more likely to be acceptable to a non-academic audience familiar with the native text.

As discussed later in this document, you can specify what form of reversification should be applied (including *None* where you really don’t want reversification); or you can leave it to the converter to make up its own mind.

## Osis2mod

In order to generate a module we use the tool *osis2mod*, which requires OSIS as input and converts this into module form.

*osis2mod* now exists in two forms. There is the original Crosswire form, and our own STEP form.

The Crosswire form is the one we have used up until recently (and the one we would have to use if we wanted to make finished modules available to third parties). However, it has certain limitations. In particular, it has a number of built-in versification schemes, and cannot cope with data which is not well aligned with one of those (or certainly cannot always cope well). And it also cannot cope with those relatively few texts where the translators have deliberately opted not to keep all of the verses in strict numerical order, or which have non-standard book ordering.

Both of these issues our own version of *osis2mod* can cope with – but at the cost, as I say, of the resulting modules being usable only within STEPBible.

As with reversification, you *can* use a command-line parameter to force a particular version of *osis2mod* to be used, but are probably better off leaving it to the processing to make up its own mind.

The STEP version *must* be used if we are using run-time reversification or if any verses or books are out of order.

# Using the converter – process overview

1. Install the converter and configure its environment. See section 3. You do this once off (but you may need to repeat portions of the job if the converter is updated).
2. Create a folder structure to hold the inputs for your text. See section 4. You do this once for each text. If you have many texts which share some characteristic (for example, many from the same text supplier such as Biblica), you may want to make some additional arrangements so that the texts can easily share common configuration information etc. This is also discussed further in section 4.
3. Set up configuration information. See section 5. You do this once for each text. You need to do at least *some* work for each individual text, but the amount of work is fortunately often quite limited.
4. Consider whether the text needs to be pre-processed and / or whether any OSIS associated with it may need to be prepared manually. See section 6. Preprocessing is concerned with ironing out the idiosyncrasies of an individual text or a set of related texts. Actually, the need for preprocessing may become apparent only after you have done a first run of the converter and examined the results.
5. Run the tool, check the outputs, make any necessary modifications, and repeat as necessary. See section 7.

# Installing the tool and configuring the environment

***You do this just once.***

* You need to have the file TextConverter.jar stored somewhere; and depending upon how you choose to run it, you may need to have it in your classpath.
* You need to have Java 18 or later installed.
* You need the *osis2mod* program. This comes in two flavours – the official one maintained by Crosswire, and our own. I believe our own will behave in the same way as the Crosswire one where we give it no reason to behave otherwise, so just that one version will in fact do.[[3]](#footnote-3)
* You need to set up an environment variable indicating the program can be found. The variable should be called *StepTextConverterParameters*, and you need it to contain the following (which you should assume to be case-sensitive):

stepStepOsis2ModFolderPath=somePath; stepCrosswireOsis2ModFolderPath=someOtherPath

(The setting for *stepCrosswireOsis2ModFolderPath* is required only if you decide you need to have both versions available.)

This environment variable can in fact be used more generally for holding configuration information. Configuration information in general, and this environment variable in particular, are discussed further in section 5.

Note, incidentally, the repeated ‘step’ at the start of the first setting. Internally, all configuration parameters have names starting with ‘step’; and here the parameter gives the location of the STEP version of the osis2mod program. Hence ‘stepStep’. My apologies – the universal ‘step’ prefix is a historical hangover which it would be kinda risky to remedy at this stage.

# Creating a folder structure

***You set this up once for each text.***

## The per-text root folder

* You need to create a separate root folder for each text.
* It doesn’t matter where you store this folder, although it may be convenient to co-locate folders for related texts (eg all texts from the same publisher) under a common ancestor folder so they readily can share configuration information. We look at this idea of shared data further in section 5.1.
* Give the folder a name like eg *Text\_eng\_XYZ*, where in this example *eng* is the two- or three- character language code for the text, and *XYZ* is the abbreviation for the name of the Bible, the two parts being separated by an underscore. My convention is to give the former in lower case, and the latter in whatever format seems to make sense. (The module name will be based upon this. It sometimes drops the language code, and, where it is retained, forces the first letter to upper case. The abbreviation is retained as-is.) The name should always start *Text\_*.
* Optionally, you may add a further portion to the folder name – eg *Text\_eng\_ESV****\_th***. See section 7.5 for an explanation of why you may want to do that.
* Very often the translators will have provided an abbreviated name for you.[[4]](#footnote-4) Where this is the case, if you have both an English and a vernacular abbreviation, use the vernacular form if it is in Roman characters, otherwise the English form. If none is supplied, you will have to make one up. It needs to be unique. There is no limit on the length of the abbreviation, but you should try to keep it short, because it is used in module names, and these appear in places where screen real-estate is at a premium.

## The content of the per-text root folder

The content of each per-text folder is standard. I’ll show below a somewhat extended version of that folder, and then give details.

**Text\_deu\_Lut1545**

\_Output

InputOsis

InputUsx

InputVl

Metadata

* The black folder is created automatically by the processing. You do not need to create this, therefore.
* You *must* create the red *Metadata* folder ahead of time. This must hold, at the very least, a configuration file called *step.conf*, and may hold other things too. Configuration information is discussed in section 5.
* As regards the blue folders, if you are starting from VL, create *InputVl* and put your raw data into it. If you are starting from USX, create and populate *InputUsx*. If you are starting from OSIS … well, you get the picture.
* If you are starting from USX or VL, the processing will itself create OSIS and store it in *InputOsis* for possible use as an alternative input on future runs, replacing any previous data there. On future runs, you can apply manual tweaks to this OSIS if you want (for example to add or modify tagging), and can request that the converter use this as its input in preference to any USX or VL. In such a case, the converter will warn you if the USX or VL data is later than the OSIS, and therefore possibly the OSIS is out of data.
* USX files are expected to have an extension of .usx. VL files must have an extension of .txt. And OSIS files an extension of .xml.
* The module zip file (and, on release runs, the repository package) are created under the \_*Output* folder. That folder will also contain other material which might be useful for debugging purposes, like the OSIS actually used in generating the module.

# Configuration and metadata

***You set this up once for each text.***

## Overview

The system is very highly configurable. In part, this is because it *has* to be – you need to be able to say different things about different texts, and you need to be able to apply different processing to them. And in part it’s because frankly we don’t really know what’s going to hit us (or certainly we *didn’t* know – and each text still seems to bring surprises).

Among other things, the metadata has to cater for:

* Controlling the conversion process.
* Describing the text and its copyright and provenance.
* Determining how USX tags are to be converted to OSIS.
* Dealing with vernacular translations of certain English information.
* Recording how references are formatted.

This implies a very large amount of configuration data. Most of this data is pretty much fixed, and for convenience resides in the *Resources* section of the converter JAR file. The configuration data for each text will need a certain amount of setting up and / or tailoring, but fortunately in general the amount of work involved is quite modest.

For details of how to set up configuration parameters refer first to \_readMeFirst\_.txt in the Resources section of the converter JAR file[[5]](#footnote-5), and then, if necessary, to the header comments of the various other files collocated with it. (Hopefully, though, you can find another existing configuration file and simply copy and tailor that.)

It may be useful when setting up your folder structure and configuration data to consider whether there is configuration data which could usefully be shared. For example, if you have a number of texts from the same publisher, they may all need to have the same details of the publisher, conditions of use, etc. It is easy to refer out from one configuration file to another – particularly to another in a closely related part of the folder structure. So you could, for instance, place the root folders for all of the texts from that publisher under a common folder, and store this shared configuration data in a file in that common folder. Then all of the individual texts could access the shared data easily, avoiding the need to duplicate it.

# Do you need to pre-process the text?

***You set this up once for each text which needs it. Pre-processing***

***for VL and USX is run automatically every time the converter runs.***

There are two places (or perhaps three if you choose to see it that way) where you might need to apply pre-processing.

## VerseLine

I’m not exactly clear whether this counts as pre-processing, but because VerseLine is not standardised, you need to inform the converter ahead of time how to parse the VerseLine data. You do this via configuration parameters, which you need to set up:

* **stepVlCommentMarker**: Defines anything used as a comment marker in the text. Blank lines and lines starting with this marker are ignored. You may leave this undefined if there are no comment lines. That’s a lower-case ‘L’ after the ‘V’ in the name – *Vl* for VerseLine.
* **stepVlLineFormat** – eg ?<**bookAbbrev**>.\*?)\.(?<**chapter**>\d+)\.(?<**verse**>\d+)\t(?<**text**>.\*)  
  A regular expression which makes it possible to extract the various parts of each line. You must define the named fields listed above (highlighted in red).
* You may also need to define **#VernacularBookDetails** – for example  
    
   #VernacularBookDetails **GEN**: Abbr: **Gn**.  
    
  #VernacularBookDetails is used throughout the system to relate long / short / abbreviated vernacular names to the corresponding USX abbreviation (ie it is also used for things other than VerseLine. The red above is the UBS book abbreviation for the text, and the green is the name as it appears in references in the VerseLine data. You need one entry for each book which appears in the VL data (or which may be created as a result of reversification). This data may be omitted if the names which appear in the VL are in fact already standard USX abbreviations.

Be aware that because there is no standard for VL, there is no guarantee that the existing processing will work with future VL texts. It is perfectly possible that the configuration details given above may not be enough for future texts, and changes to the processing will be required.

## USX

USX is complicated, and people get it wrong. Even if they get it right, it isn’t always what we want.

In fact, certain *very* common issues are corrected within the converter itself. In general, however, the converter doesn’t sort out the text because – just as with VL – each text is likely to have its own unique issues.

There are a number of options for addressing the shortfall.

You *can* address issues manually if you want – ie just edit the inputs to correct them before running the converter. If you do that, though, your edits will be lost if the translators issue a new release.

Alternatively, you can write your own automated pre-processor and use it to modify the data before you supply it to the converter.

And / or, you can supply fragments of XSLT via configuration parameters *stepXsltStylesheet* and *stepXsltStylesheet*\_*Gen* etc. The latter gives transformations to be applied to a particular book. If you have both a book-specific value and the general one, the book-specific one is applied to that particular book, and the general one to any books which lack a book-specific sheet.

The value assigned to these parameters can be either a complete XSLT stylesheet or a collection of *xsl:template* chunks. I’d recommend the latter, because the processing will then automatically take care of namespace-related issues for you. Thus something like the following is perfectly acceptable:[[6]](#footnote-6)

stepXsltStylesheet\_Psa#= \

<xsl:template match="para[@style = 'qd']"> \

<xsl:copy> \

<xsl:attribute name="style">d</xsl:attribute> \

<xsl:attribute name="\_X\_suppressValidation">y</xsl:attribute> \

<xsl:apply-templates select="@\*[not(local-name()='style')]|node()"/> \

</xsl:copy> \

</xsl:template>

## OSIS

As an alternative, you can supply OSIS as input.

If tweaks have to be applied to the OSIS, you will have to handle that before running the converter. The converter does actually apply *some* tweaks of its own in order to overcome issues where STEPBible does not format standard OSIS correctly (see section 1.3), but in the main, modifications will be down to you.

# Running the converter from the command line

***You may need to do this a number of times for each text  
if issues are reported and you need to fix them.***

At the time of writing, the converter can actually be used for three entirely separate purposes.

The main one, naturally, is to generate a module and / or a repository package from the various inputs. This is discussed in section 7.1.

But you can also use it to compare the text with the various built-in versification schemes supported by Crosswire’s version of *osis2mod*. This would be a useful thing to do if you wish to make a module publicly available – to do that, you *have* to use the Crosswire version of *osis2mod* and need to choose a versification scheme which the text fits reasonably well. This is discussed in section 7.2.

And you can use it to check whether a given module was built from a given set of inputs. This is discussed in section 7.3.

Limitations in the library I am using to parse command line parameters make it difficult to tailor the list of parameters according to which of these you wish to run. I therefore always accept everything, and ignore any parameters not relevant to a particular run.

## Generating a Sword module

java -jar PathToJarFile\TextConverter.jar  
 **-rootFolder PathToFolderToBeProcessed**  
 -runType …  
 -startProcessFromOsis  
 -updateReason …  
 -reversificationNotes …  
 -manualOsis2mod …  
 -forceOsis2modType …  
 -dbgSelectBooks …  
 -dbgAddDebugAttributesToNodes  
 -dbgDisplayReversificationRows …  
 -checkInputsAgainstPreviousModule  
 -evaluateSchemesOnly  
  
(To obtain help information, run the program without command-line parameters, or with the parameter  
–help.)

The blue parameter is required regardless of which purpose you are using the converter for. The green parameters are relevant when generating a module. The red parameters select between the other uses of the converter as discussed above.

**rootFolder** is the path to the root folder for the given text (for example the *Text*\_*eng\_XYZ* folder mentioned earlier). If it’s an absolute path, that’s what’s used. If it’s relative, the system attempts to locate it relative to the current working directory. If that doesn’t work, it looks for the setting stepTextConverterDataRoot in the environment variable mentioned in section 3 and tries looking for it relative to that.

**runType** may be *Release / MajorRelease / MinorRelease* (generate a release version of the module); or *EvalOnly / EvaluationOnly* (generate a version for evaluation only). Version numbers and history information are updated only on release runs. You can force the question of whether this is a major or minor release using *MajorRelease* or *MinorRelease*. If you just use *Release*, the processing will decide for itself whether this is a major or a minor release. See the discussion of version numbering in section 7.4 for an explanation of how the choice affects version numbers and history information.

**startProcessFromOsis** tells the converter to ignore any VerseLine or USX data, and start working from the data in the *InputOsis* folder. (The converter will, however, raise a warning if the VL or USX data post-dates the OSIS, on the grounds that this may mean you’re working from an old copy.)

**updateReason** is provides an explanation for a new release. It is not always required: if the update arises from a revision to the source information, it may well be that the text suppliers have given an explanation, and particularly if you are working with DBL, the processing may be able to pick it up from the DBL metadata automatically. Refer to section 7.4 for more details.

**reversificationNotes**: None / Basic / Academic. Determines what kind of reversification-related footnotes are added to the text. *Basic* and *Academic* differ in the number and level of details in the footnotes. This parameter is optional, and defaults to *None*.

**manualOsis2mod**: This may be useful particularly if running the converter from within the IDEA IDE. Normally running *osis2mod* from within the converter works perfectly well, even when you are running the converter itself within the IDE. And the processing is indeed set up to run *osis2mod* automatically in this way. However, latterly when processing ESV (a particularly large OSIS file – around 32Mb – a fact which I presume may be significant), *osis2mod* would just hang. If you pass *manualOsis2mod*on the converter command line, the processing stops at the point where it would otherwise run *osis2mod* itself, copies to the clipboard a command you can use to run *osis2mod* in a plain vanilla command window (on Windows, that’s just a command window, not a Powershell window), and then lets you tell it when *osis2mod* has completed, at which point it carries out the processing which it would *normally* perform after *osis2mod* had run.[[7]](#footnote-7)

**forceOsis2modType** can assume the values *Step* or *Crosswire* (not case sensitive). The converter can use either the Crosswire osis2mod to create modules or a STEP bespoke version. This forces the choice. If not specified, the converter makes the decision: if the versification of the text is close to that for KJV(A), and if no verses and / or books are out of order, it uses the Crosswire version, otherwise the STEP version. You need to be careful if forcing things, however. If you force the use of the Crosswire variant, but the text contains verses which the selected *osis2mod* versification scheme does not support, the resulting module will contain errors. (This assumes, of course, that you have set things up in such a way as to have both available. If not, you should have the STEP version available, and that’s the version which will be used.)

**dbgSelectBooks:** When debugging, you may be able to speed things up by limiting the books being processed on a given run. This parameter lets you do this. The value can be eg *Psa* (to process just Psalms); *Mat,Mar,Luk* (to process those three books); <*Lev* (to process books before *Lev* in the standard ordering); >=*Lev*, etc.

**dbgAddDebugAttributesToNodes:** The processing is set up to add extra attributes to nodes to aid in debugging. This setting determines whether this happens or not.

**dbgDisplayReversificationRows:** lists the rows of reversification data which apply to this text. The argument to this option determines where this output is sent. May be *No*, in which case no output is produced; anything containing *screen*, in which case output is sent to the screen; or anything containing *file*, in which case output goes to debugLog.txt in the root folder for the text. You may include both *screen* and *file*, in which case output goes to both places. By default, output is displayed at the time it is available. Include *deferred* in the argument to have it generated at the end of the run (at the end of the run, the output is guaranteed not be interleaved with other information). This parameter is optional, and defaults to *No*.

**dbgAddDebugAttributesToNodes:** The processing is set up to add attributes to certain nodes which may be useful for debugging purposes, but at the expense of producing larger files. This setting determines whether these addition attributes are added or not. This parameter is optional and defaults to *No*.

## Evaluating versification schemes

java -jar PathToJarFile\NewConverter.jar  
 -rootFolder PathToFolderToBeProcessed  
 -evaluateSchemesOnly

**rootFolder** is as discussed in section 7.1.

This command evaluates the extent to which each of the versification schemes built into *osis2mod* supports the versification actually present in the raw USX files. It outputs a score table to stdout, along with details of how to use the scores to select the scheme which best fits.

If you are using reversification to restructure a text at conversion time (see section 7.5 for a discussion of reversification), or are using the STEP version of *osis2mod* to restructure the text as necessary at runtime, then you do not need this information. If neither of these applies, however, you will need to use this information to select a suitable versification scheme from those supported natively by *osis2mod*, and must then set the stepVersificationScheme configuration parameter accordingly before generating a module.

Note that texts relatively seldom fit any ‘standard’ scheme exactly, and if you choose one of the built-in schemes, *osis2mod* may itself restructure a text which does not precisely fit that selected scheme. This is probably undesirable. If you want to work this way, though, then in general, if no scheme fits exactly, you should choose one where the scheme expects verses which the text does not supply, rather than one where the text has verses which the scheme cannot accommodate.

## Checking the inputs for a module

java -jar PathToJarFile\NewConverter.jar  
 -rootFolder PathToFolderToBeProcessed  
 -evaluateSchemesOnly

**rootFolder** is as discussed in section 7.1.

When the converter creates a module, it stores SHA256 digests for each of the input files in the Sword configuration file which forms part of the module. This option compares these digests against the inputs as they currently stand, to determine whether the module was built using those inputs. (Where you have VL or USX available, but opt to start from OSIS, it is the OSIS whose digest is retained.)

## Version numbering and history information

Crosswire stipulates that version numbers should be of the form *<major>.<minor>* – eg *1.2*.

It is actually rather difficult to determine in general what should constitute a major update and what a minor one, because we would be dependent upon text suppliers to make this clear, and they do not always do so (indeed, sometimes we don’t even have version numbers from them) – and even if they do, they may differ as to what is a major update.

I have therefore taken an easy way out: we treat as major any update occasioned by the release of a revised version of the source package by the text suppliers, no matter what the revision may have been; and we treat as minor any release occasioned by a revision to the conversion software.

There is absolutely no reason why this should be appropriate – a change to the source package may have zero impact, and a change to the software may have a huge impact – but at least it’s easy to be consistent.

If you need to force a particular change to be viewed as major (or as minor) when it would not naturally be seen as such, you can use the *runType* command line parameter as discussed in section 7.1.

History information has to end up in the Sword configuration file. However, this file is regenerated on each run, and therefore cannot be used as a long-term repository of the information. In view of this, I update the step.conf file to hold the history information. If this information isn’t what you want, you are free to edit it in just the same way as any other configuration information, so long as you retain the correct format. However, a) don’t move history information out of step.conf and use the ‘include’ mechanism to store it anywhere else; and b) keep it as plain text – don’t rely upon any of the built-in configuration mechanisms to construct it out of other things.

History and version information is updated only on release runs.

It is up to you to maintain *Obsoletes* information manually in step.conf in this form:

**copyAsIs=Obsoletes=**abcXYZ

The stuff in boldface should always appear exactly as it appears here, and at the end you should give the module name of any existing module which this one replaces. You don’t *have* to have any of these lines; but if you do, you can have as many as you like, to cater for the possibility that a new module obsoletes a number of existing ones. There is no need to include a line for the module itself: if you are constructing, say, deuHFA, it is automatically assumed that this will be replacing any previous copy of this module.

## Module naming / reversification / use of STEP’s own version of *osis2mod*

Sadly (in terms of making this even vaguely straightforward) all of the issues named in the title of this section are inter-related. (See also sections 1.5 and 1.6.)

**To reversify or not to reversify**

You can force conversion- or run- time reversification, or you can specify that no reversification is to be applied … or you can leave it to the processing to decide. I recommend this last option unless you have very good grounds for forcing the issue. (‘Good grounds’ might be that you know the text is already NRSV(A) compliant, or that you are generating a module for some third party and therefore don’t need it to be NRSV(A)-compliant.)

If left to its own devices, the converter will never apply conversion-time reversification. But it *will* apply runtime reversification if the text is a long way adrift of NRSV(A). (More specifically, it will definitely apply runtime reversification if the text contains verses which NRSV(A) does not accommodate.)

**Crosswire *osis2mod* or STEP *osis2mod***

I mentioned earlier that using our own version of osis2mod is probably preferable, because it will do everything that the Crosswire version does and more, and is more likely to be up to date.

If you happen to have both versions available … STEP’s *osis2mod* is required if *runTime* reversification has been selected. It is also required if the text has out-of-order verses or books. Other than for these two cases, Crosswire’s version of *osis2mod* will work.

If you are using only STEP’s version, and need to generate a module which can be made available to third parties you need to specify in step.conf one of the built-in schemes supported natively by osis2mod, and you need to suppress reversification processing. (You will also need to suppress encryption.)

**Selecting reversification and osis2mod options**

You can attempt to force either of these, using the configuration parameters *stepForceReversificationType* and *stepForceOsis2modType*.

In both cases, you can force a *more* stringent type than is required (*runTime* or *conversionTime* is more stringent for the reversification type than *None*, and *Step* is more stringent than *Crosswire* for the *osis2mod* type).

If you are forcing something *more* stringent than the above rules require, the processing will issue either a warning or an information message, but it will respect your request. If you attempt to force a *less* stringent type than is required, the processing will abort with an error.

Or you can simply not force the issues at all, and leave it to the processing to decide.

**Module naming**

Module naming is derived in part from the name of the root folder for the text, and in part from the nature of the processing being applied to it.

A module name looks something like:

<langCode><abbrev>[\_suffix] eg deuLut1545\_th

*langCode* is the 3-character language code for the text, first character in upper case, remainder in lower case. It is omitted on English texts and on ancient Hebrew and Greek texts.

*abbrev* is the abbreviated name for the text, and is taken from the root folder name. It is used exactly as specified there.

*suffix* is optional, and is supported only because some modules were given this further suffix at some point in the past and it is needed for backward compatibility.

The above describes the situation for release runs. If you are generating an evaluation-only module, the name reflects this fact, and also includes a date stamp, so that you can retain multiple different copies if necessary – eg *DeuHFA\_EvalOnly\_0927T1908.*

## Checking the outputs

Each run of the converter generates two log files – *converterLog.txt* and *osis2ModLog.txt –* in the per-text root folder. The latter contains messages generated by the *osis2mod* program, and the former contains messages reflecting the overall conversion process. *converterLog.txt* contains copies of the most important messages from *osis2mod*, and so normally you need refer only to *converterLog.txt*. The converter itself gives an indication on stdout of whether errors or warnings have been issued, and therefore whether you need to refer to these files.

## The Sword module

The Sword module is created as a zip file in the *\_Output* subfolder. Its name reflects the module name as discussed in section 7.5.

On release runs, a repository package is generated, also in the \_*Output* folder. This package contains the module zip file and various of the input data (for example the OSIS used to create it).

## Additional information: the TextFeatures folder and the enhanced Sword configuration file

The processing creates a *TextFeatures* folder within the Sword module folder structure, and stores two files within this, in case they prove to be useful.

*textFeatures.json* summarises things like which USX tags the text uses, whether it contains tables, etc.

*vernacularBibleStructure.json* indicates whether the text contains OT books, NT books, DC books, full or partial OT and NT, and which books it actually contains.

And the Sword configuration file is also enhanced with a lot of header comments in stylised form which could perhaps be used for the purposes of automated processing in support of administering the texts (for example, to identify texts whose licences are due to expire shortly).

A separate Kotlin tool – *AdminDataExtractor* – is available to extract this administrative data from collections of modules.

**Maintenance guide**

# Philosophy and implementation

## Overview

The converter is a large collection of Kotlin code which takes Bible texts in a variety of formats, creates from them an ‘equivalent’ OSIS representation, and then uses the external program *osis2mod* to turn this into Sword modules.

The input formats currently supported are USX, VerseLine (VL) and OSIS (and also, by implication, USFM, because the UBS Paratext tool can be used to convert this into USX).

The overall flow of control of the converter is shown below.

|  |
| --- |
| External OSIS  USX  VL  Module Repository package Encryption data etc  External OSIS  Internal OSIS |

A given text must, when first processed, have at least one *InputXxx* folder. (It can also have two *InputXxx* folders, so long as one of them is *InputOsis*. In this case, by default processing starts with the non-OSIS folder unless you specify otherwise.)

USX and VL are converted to externally-facing OSIS (see section 1.2). This is OSIS which should be acceptable to a third party, were we in a position to make it available (subject to the considerations discussed in section 1.2). This version of the OSIS is stored in the *InputOsis* folder as a possible alternative input for future runs (where, for instance, we need to tweak the tagging and are set up to do this only to OSIS). It is not actually stored there until the end of the run though – I don’t store stuff until I know things have worked. Any previously existing pure OSIS is deleted at this point.

Where we are taking OSIS as input, I assume that this is in external OSIS form to begin with.

The pure OSIS is then modified in various ways (for example to apply reversification, to expand elisions, and to apply modifications needed to sort out STEP’s rendering), before being fed to *osis2mod* to create the module. This modified OSIS is of no long-term value. It is retained for debugging purposes (retained until the next build, that is, when it will be overwritten), but should not be used for anything else.

And then finally I generate the various supporting data (features summary, encryption data, etc) and create the module, the module zip file and, on release runs, the repository package.

## Code structure – background

A note on history. Previous versions of the converter were all based upon USX – USX was essentially the only form of input for which we catered, and so all of the manipulation (expansion of elisions, reversification, etc) was applied there. The move to OSIS has meant reorganising things so that this manipulation is applied to OSIS rather than USX. At the same time, I recognised that in fact much of what I was doing could equally well be applied in either place – the two might know some kinds of nodes by different names and use different names for their attributes, but allowing for that, a lot of the actual processing would be pretty much identical.

Where I felt it might be useful, therefore, I have created a standard framework to do the work – a framework which doesn’t actually care whether you are working with USX or OSIS. You tell it which applies at the time you instantiate it, and it should then just work regardless of which protocol you are handling. This comes at the cost of a slight increase in complexity, but it does give us the flexibility to revert to working with USX again should that ever become necessary.

Having said this, you should not assume that the code as it currently stands will work *correctly* with USX. Hopefully in most respects it will, but at the time of writing we have no plans to revert to USX, and therefore the USX side of things has not been tested at all.

Just in case you *do* have cause to revert to applying the bulk of the processing to USX, Utils, Support and SE items should hopefully carry through pretty much as-is. The PE items will definitely need attention, but even there, the overall flow of control may well be much the same.

## Code structure

The code is split into a number of different packages.

### Main

The main level includes *Main* (the main program); and *ProcessingController*. The former does little more than pass control to the latter, and provide a backstop to catch and report exceptions. The latter handles command-line arguments, arranges for configuration data to be read, works out what processing steps are required, and runs them.

### ProcessingElements

This includes all of the main elements of the processing activity – the steps which *ProcessingController* selects and runs.

There are three *PE\_Phase1\_FromInputXxx* classes, each of which handles a particular type of input (VL, USX or OSIS).

This is followed by *PE\_Phase2\_ToInternalOsis* which is concerned mainly with converting the external OSIS created by the previous step to the internal OSIS needed in order to generate the Sword module.

*PE\_Phase3\_To\_SwordModule* runs *osis2mod* and then zips the results up to form a module.

*PE\_Phase4\_To\_RepositoryPackage* creates a repository package and (where the original input was not OSIS) saves the generated OSIS to *InputOsis* for possible future use. It does this only on release runs, however.

And *PE\_InputVlInputOrUsxInputOsis\_To\_SchemeEvaluation* is used on runs whose sole purpose is to evaluate how well the data fits with the various versification schemes built into *osis2mod*.

### Subelements

These are things which are used by the processing elements described in the previous section. Their names kinda describe what they do; beyond that, the head-of-class comments go into much more detail. All of them could work equally well on either USX or OSIS (I *think* …), and take an *X*\_*DataCollection* instance as argument to their constructor (see section 8.3.6) to provide them with access to their inputs and also the tailoring necessary to enable them to work with either USX or OSIS as appropriate.

### support

This contains what are essentially utility methods – things not tied directly to the purposes of the main application, but which might be of wider utility (eg things to delete files, to manipulate strings, etc). This probably could and should be turned into a separate standalone library, but I’ve never managed to fathom how to get IDEA to work with that.

### osisinputonly / usxinputonly

Various bits and pieces specific to a particular kind of input.

### utils

This contains utility methods and classes which (unlike ‘support’ – section 8.3.4) are very closely tied to the application. In general they are either agnostic to whether USX or OSIS is being processed, or are tailored via runtime parameters.

Two of them warrant more detailed description:

* *X*\_*FileProtocol*: This contains the information, widely used through the system, which enables other classes to work either with USX or with OSIS (objects *UsxFileProtocol* and *OsisFileProtocol* are derived from *X\_FileProtocol*). For instance, this tells the system how the particular protocol represents a book node, how to create footnotes, etc.
* *X\_DataCollection*: Gathers together a lot of information about the inputs, along with a suitable bundled instance of either *UsxFileProtocol* or *OsisFileProtocol*. This means that most of the system works off one of these. Within the Globals file are a number of instances based upon this – *ExternalOsisDataCollection* (holds the output of Phase 1 processing); *InternalOsisDataCollection* (holds the temporary OSIS data which will be fed to *osis2mod*); *UsxDataCollection* (holds input data during initial USX processing).

# Tools etc

There is, in the git structure, a folder called *Support*, containing various items which are either directly useful, or which may prove to be useful at some point in the future. The main things which I use regularly are:

* **bibleStructure.xlsx**: Gives book names, number of chapters per book and number of verses per chapter for USX and OSIS.
* **reversificationRaw.xlsx**: A copy of the extended reversification data. This spreadsheet isn’t used by the processing, but it is useful for reference when debugging, because it is more readable. Or it’s useful so long as you keep it up to date by copying data from the website and pasting it into the file on a regular basis. (It needs to contain the expanded version of the data, and all comments and blank lines should be removed from it.)
* **stepTextManagement.xlsm**: An attempt at drawing up a list of what texts we have, what texts are available, and what texts we are working on – although at the time of writing this is significantly out of date and incomplete.
* **protocolDetails.xlsm**: Records information about USX and OSIS tags, and lets you generate from this information code to be used within the converter.

The *Support* folder also contains other miscellaneous odds and ends, third party documentation, Crosswire files, etc.

# OSIS conversion

## Non-compliance

To make our modules available in the Crosswire repositories, the OSIS we generate would have to be fully compliant with the OSIS standard. Unfortunately in general they will not be. For more information about some of the issues and the reasons for non-compliance, see section 1.2.

## Conversion philosophy

To generate a module, we have to create OSIS (unless we have been given OSIS as the input). The process is as follows:

* If starting from VL or USX, I generate OSIS, which I also save in the *InputOsis* folder in case we want to do a later run to take on board manual tweaks applied to the OSIS. This is the *external OSIS* discussed in section 1.2.
* If starting from OSIS, it is convenient to imagine a similar process, in which I turn that OSIS into a form more immediately amenable to processing . Here this will be mainly a case of ironing out any idiosyncrasies which may arise if the OSIS was supplied to us by third parties. In this case I do *not* replace the input OSIS with this revised version – the starting OSIS continues to be available to us until we manually replace it or do a new run starting from VL or USX.
* I now convert *externally*-facing OSIS to *internally*-facing OSIS; and then I use this to create the module. This version is essentially throw-away: once the module has been generated we no longer require it (unless we want to keep it for debugging purposes).

As regards tag conversion, asfar as possible, I have followed Appendix F of the OSIS reference manual, which contains SIL’s recommendations for mapping USX tags to OSIS tags. However, given that in many cases there really *is* no obvious mapping, this is at best somewhat *ad hoc*, and I have departed from this wherever it seemed expedient to do so. Most mappings are actually configurable on a per-text basis[[8]](#footnote-8) (you just need to override definitions in *usxToOsisTagConversionsEtc.conf* in the JAR’s *Resources* section), so that you could, for instance, map *char:add* to one OSIS construct for one text, and to another construct for another. I would recommend reconfiguring them only *in extremis*, though.

## STEP OSIS

Some of the things below are really *extremely* arbitrary, and represent workarounds for what appear to be bugs in downstream processing and rendering (bugs which really ought to be fixed, rather than hack the module, but at present it doesn’t look as though that will be feasible). Others have a rather more compelling rationale in terms of what we are trying to achieve.

* Verse ends may be moved, and tables may be altered, as discussed in various places in section 1. I also replace start-of-chapter canonical heading tags with formatting markup to avoid cross-boundary issues (and end-of-chapter headings too, but for a different reason: these are mentioned below).
* Elisions are expanded out into individual verses.
* ‘Speaker’ tags are replaced with formatting markup. STEP does render speaker tags, but we don’t like the way it does it.
* Acrostic tags (as paragraphs and as span-type) are also rendered in a way we don’t like, so I replace the tag with formatting markup.
* Selah tags aren’t handled well either, and again are replaced with formatting markup. In theory, the word is supposed to be rendered right-justified on a line to itself. I have found no way of achieving this: the best I can do is to leave the word on the line where it appears, but italicise it. As a further source of confusion, Selah is given by a char tag in USX, but is a para tag in OSIS.
* In one recent text, where a comma preceded a <note> tag, the comma was occasionally dropped. To get round this I always insert an apparent entirely redundant <hi type='normal'/> before the <note> in such cases.
* There is a similar issue with poetry tags – on one text, poetry lines were arbitrarily being dropped (some appeared in the output, some did not, and there was apparently no pattern to it). A similar expedient to that in the previous bullet point seemed to fix this: I simple include <hi type='normal'/> before any poetry tag.
* Identical verses: This came up in the context of elision processing, and I can’t immediately think it will be an issue elsewhere. Anyway, with elision processing, you tend to end up with runs of empty verses, and for consistency’s sake, you really want them all to look the same (ie to have the same content, such as an ellipsis or a dash). However, somewhat bizarrely, if you have a consecutive block of verses all with the same content (or all with no content), some of them are suppressed (and yes – even more bizarrely, it’s only *some* of them which are suppressed). To get round this, I am adding a no-effect char-type markup on alternative verses of this kind, so that no two adjacent verses have the same content.
* Blank lines: It has latterly become apparent that something somewhere can get screwed up if blank lines appear in the text in certain places – either via USX para:b or via an empty para:p. Blank lines at the very end of a chapter cause the last verse number to come out *after* the text of the verse (and I therefore automatically remove at least para:b from this situation to avoid this). But I have also noticed in at least one case that a blank line actually caused a *later* verse to be dropped in its entirely (and not even an adjacent verse). This is clearly worrying, but at present I have no real handle on what is going on. Plus also more recent experience suggests that a para:b introduced not to split an existing line but purely to force a blank line may be ignored. If you really want the blank line to appear, the line actually needs to have something on it (like &nbsp;).
* Blank lines at the ends of chapter are not only pointless but also result in rendering errors, with verse numbers coming out in the wrong place. I therefore delete terminal blank lines.
* There are a few places where canonical titles are situated at the ends of chapters. If left as such, weird things happen (eg verse numbers being misplaced or verses being moved to the next chapter). I therefore convert them to formatting markup.
* Cross-references may be changed to plain vanilla footnotes if they target verses which do not exist in a particular text (eg a reference to the OT from an NT-only text), or if they appear to be invalid.

## A special note on tables

As suggested above, tables which contain verse markers are a particular problem (except possibly if, throughout the table, the sid for each verse appears in the same cell as the eid, but I have yet to encounter a table like that). Any attempt at all to retain them in this form is likely to result in *osis2mod* complaining or – worse – there being no complaint, but things coming out entirely wrong. And of course verses which have markup running across their boundaries are a big problem for reversification (the one redeeming feature here being that I believe it unlikely that those portions of a text most likely to be subject to reversification will be formatted as tables).

To address this, I retain the table markup, but remove the verse markup, creating an elision instead. Thus if the table originally spanned vv1-10, we end up with vv2-9 empty, and the entire table in v1. This retains the tabular appearance, but at the cost of the individual verses no longer having their original content (with unavoidable knock-on implications for added value such as verse vocabulary, interlinear, etc).

As regards table tags, the OSIS reference manual admits that support for tables is somewhat half-hearted, and also suggests that implementations will need to add their own ‘*x-*’ attributes in support of processing (something which surely of itself limits the usefulness of making OSIS modules available to third parties). STEP does precisely this, with attributes like *x-simpleTable*, for instance. Unfortunately, though, I have been unable to find any documentation as to what attributes STEP requires or supports. I can only say that with the text I have handled to date, we have successfully handled *style='tr'* on USX *row* tags, and things like *<cell align="start" style="tc1">* and *<cell align="end" style="tcr2">* on USX cells – so the OSIS we are generating for these is clearly appropriate. The USX *align* parameter presumably dictates alignment (and is actually the one standard attribute defined in the manual), so I am not sure why ‘r’ is needed in the second example, since presumably it, too, refers to right alignment. I have a feeling there are quite a number of other possibilities, but I have no idea what they are.

STOP PRESS: We have recently come across an alternative formalism for tables, in which table- and row- tags are not used. With a two-column table, the content of each entry in the left-hand column is enclosed in <cell>, and the right hand column follows as plain vanilla text. This may work only with two-column tables (need to investigate), and to my mind produces something which is not well enough formatted to be useful, but we’ll have to see.

## A special note on canonical headings

Canonical headings at the start of chapters are particularly problematical.

* Some texts do not have any heading tags (but need to have them courtesy of reversification).
* Others have them, but they may be marked up in various different ways …
* There may be more than one heading tag.
* The heading tag may contain one or more verse markers, or it may contain none.
* If it does contain a verse marker, there may be canonical text before that marker or there may not.
* Similarly any verse in the heading may end at the end of the heading, or it may continue after it.

In fact the rendering of ‘pukka’ canonical headings on our existing texts is not particularly complicated – it appears, actually, to involve no more than italicising the content and placing it on a line by itself. Fortunately this can be achieved in a manner which avoids cross-boundary markup altogether, so I always apply that change.

If the text is amenable to using the Crosswire *osis2mod [[9]](#footnote-9)* (ie if it is NRSVA-compliant, or deviates only in a ‘good’ way, and if the books, chapter and verses are all in ‘normal’ order), that’s all I do: I make the assumption that the text was previously marked up in an appropriate manner, and that my changes will not have undone that.

If the text needs to be run through our own *osis2mod* (ie the bracketed conditions in the previous paragraph do not apply), I again leave the text as it now stands, and make the assumption that in passing details of reversification Move’s and Renumber’s to that, any issues will be taken care of.

If the text is one of the few public domain ones which we might want to restructure during the conversion process to achieve NRSVA-compliance, I remove any existing markup altogether, and rely upon reversification processing to reinstate it appropriately.

## A special note on cross-references

Cross-references are complicated, in that both USX and OSIS seem to have two different ways of representing them (although needless to say, there is no direct mapping between the two).

USX *ref* tags are generally easy. The only real complication comes where reversification changes verse numbering. In this case, I update the *loc* parameter (which is in USX format) to point to the right place. At one stage I also felt the need to update the content of the *ref* tag content (which gives the same reference in vernacular form).[[10]](#footnote-10)

USX *char:xt* tags are a lot more awkward. They *may* have a *link-href* parameter which is the equivalent of the *ref loc* tag, but very often they don’t. They may already contain a *ref* tag, but again often they do not, in which case we have to generate one. Or worse, we may have to generate more than one, because *char:xt* can contain reference *collections* where *ref* cannot.[[11]](#footnote-11) And worse still, they will be in vernacular form, which means there is nothing we can do with them unless we have the necessary information to enable us to parse and create vernacular references – see further discussion below.

**Cross-reference errors**: We have seen quite a number of texts in which the cross references have not been checked. This means we have to cater for a number of different kinds of errors. If we have a reference which points to a part of the Bible which the text does not contain (for example a cross-reference in an NT-only text which points to the OT), I convert the cross-reference to plain text, but do nothing else. If the cross-reference is syntactically invalid, or points to somewhere which looks right, but in fact is wrong (eg Jn 3:999), I also tend to be fairly forgiving, on the grounds that it is probably still worth creating a text even if one or two references don’t work. Where we have more significant problems is when it comes to parsing vernacular references …

**Vernacular references**: As explained above, when processing *char:xt* we have to be able both to parse and (probably) to create vernacular references. When handling *ref’s* which target verses subject to reversification, we may also need to be able to create vernacular references. Both of these require us to know how vernacular references work, and in general we will not do so. If vernacular references happen to follow USX format, the processing will work; if not, it will need information about the structure of vernacular references, and quite aside from the fact that setting up a full description, although do-able, will definitely be painful, there is the more significant issue that obtaining the necessary information in the first place will be difficult.

# A note on debugging

When processing an entire text, it may well become apparent that there is an issue with some particular scripture file which warrants further investigation; and you may then wish to home in on that one book without having to wait while the processing deals with other books which may precede it in Bible sequence.

At the top of the main function in Main.kt is a statement Dbg.setBooksToBeProcessed("…"), which can be used to limit processing to an individual book, to a list of books, to a range of books, etc. Alternatively, you can supply this same information via the command-line parameter *dbgSelectBooks* – see section 7.1.

# Gotchas and arcane information

Here, in no particular order, are some of the issues which I have encountered, and which you should bear in mind when making changes.

**Style sheets**: Styling may well be an issue, particularly when dealing with non-English texts (and perhaps more particularly, with texts which use non-Roman characters). I do make provision for vernacular translations of the *text* of things like standard footnotes (although it may well be difficult to obtain the necessary translations); but it would be very useful to arrange for *formatting* to be appropriate to the language and its conventions as well. Material from DBL actually comes with a styles file which gives this information, but unfortunately there appears to be no way in which we can actually make use of it.[[12]](#footnote-12)

**Special characters**: Don’t be tempted to use XML characters of the form &#...; in any text. Or rather, as I recall, you have to have them as &amp;#...; . Don’t use any three-byte Unicode characters either (which sometimes might inadvertently be introduced by copying and pasting them from other applications into the USX text, even without recourse to &#...; markup). These aren’t rejected, but cause really weird problems.

**RTL**: At one time, RTL texts had to be rendered in verse-per-line format, or else the text came out in the wrong order.[[13]](#footnote-13) I am not sure whether this is still an issue. I did have formatting in place to take care of it automatically, and also allowed you to force the issue if you need to using the *stepForceVersePerLine* configuration parameter.

**Cross-references**: One thing to look out for. I have been told that if *osis2mod* is confronted with a purported reference which in fact is invalid, it tends to replace it simply by a reference to Rev 1. If you see spurious references to Rev 1, therefore, it is probably worth looking for invalid references in the raw text.

# Intellij IDEA

## Run configurations

When you create a run configuration, you have to indicate the main class. Ostensibly, IDEA locates this for you. Latterly it has stopped doing so. To get round this, use the Project option to select the main class manually, and then drill down through src/main/kotlin to locate and select the file:



## Artifacts

You need to create and build an ‘artifact’ in Kotlin terms in order to get a working JAR file, which you can use either standalone in a command line, or as the program to be run under control of my admin tools.

**Don’t forget to rebuild this each time you make any meaningful change to the code.**

Unfortunately, this folder does not turn up in the JAR by default – you need to change the configuration to make it happen: click the ‘+’ button, select *DirectoryContent*, locate the resources folder, and then add it.



Something at some point appears to generate a META-INF/MANIFEST.MF file within the resources file, and the latter should name your main class. (I’m not sure what causes this, but so long as it does, and so long as the class is named correctly, we’re on to a winner.)

Next, you have to arrange for whatever contains this META-INF folder (which I am assuming, as I say, to be the resources folder) to appear at the top of the list of entries. To this end you need to ensure alphabetical ordering is turned off, and then use the up-arrow key to move the folder to the top. There was also a suggestion that you might need to move the main class just below this; I’m not sure whether it’s necessary, but there’s no particular harm in doing it.



So far this is something you need do only once, when configuring the artifact.

Once you have built it, you can find the generated JAR a few levels down under ‘out’ in the project window.

Before it can be used, you need to go to the containing folder, open the JAR with 7-zip, go to the META-INF folder, and delete any files with names like \*.RSA or \*.SF (I think one other was mentioned on the web, but I can’t now recall what it was – currently I’m only seeing \*.RSA and \*.SF). You need to delete these files every time you generate a new version of the artifact.

## Bugs

As of 23-Sep-23, *isEmpty* on optional values doesn’t seem to work. If you give it in this form, you are told to convert it to a function call – *isEmpty()*. If you give it in that form, you are recommended to use the original form. And both give a syntax error. *!isPresent* seems to work, though.

**\* End of document \***

1. Personally, I don’t think using OSIS as the basis of changes is ideal. If we have USX available to us, using OSIS as input means we have two different inputs and have to worry about whether we have chosen to work with the right one. On the other hand, given that OSIS is the one factor common to all modules, regardless of the form in which they are supplied to us, I can see that there are some benefits in sticking with OSIS. [↑](#footnote-ref-1)
2. This is not something I imagine we will do commonly. It is used to make a text conform to NRSVA structure, which is required if STEP’s added value features are to work properly. However it generates a text which diverges in some areas quite significantly from the original. Licence conditions are likely to preclude this on all but public domain texts, and even there we will probably limit ourselves to texts likely to be of interest mainly to an academic audience who will understand the need for the restructuring. [↑](#footnote-ref-2)
3. If you need the Crosswire version, you need to be aware that only the Linux version is guaranteed to be kept up to date. A windows version is available at <http://dl.thehellings.com/sword-utils/> but it is very elderly. [↑](#footnote-ref-3)
4. If you are working with DBL texts, this will usually appear in the DBL *metadata.xml* file and there is processing which can pick it up automatically from there if you choose to use it. [↑](#footnote-ref-4)
5. You can access the content of a JAR file using a standard zip tool. [↑](#footnote-ref-5)
6. If you have a multiline configuration parameter like this one, you need to end each line but the last with a backslash as here. The backslashes are part of the STEP configuration language, not of XSLT. [↑](#footnote-ref-6)
7. Whether this option would also be useful when running the converter direct from the command line I am not sure: I suspect not, but I have little experience of working that way. [↑](#footnote-ref-7)
8. A few mappings, however, cannot be altered via configuration changes, because they involve actions too complicated to express using configuration parameters, and their processing is therefore hard-coded. [↑](#footnote-ref-8)
9. Or would be amenable to this. As mentioned previously, it may be better to use only our own version. However, it is possible to make this behave like the Crosswire version by telling it to use one of its built-in versification schemes. [↑](#footnote-ref-9)
10. At one point I felt this was a useful thing to do. Latterly, DIB has suggested it may not be necessary. In any case, the number of texts to which we apply conversion-time reversification is likely to be extremely limited, and this revamping of cross-references is an issue only for such texts. Indeed at the time of writing, conversion-time reversification in its latest incarnation has not even been tested. [↑](#footnote-ref-10)
11. There seems to be an urban myth that STEP can handle individual cross-references which are themselves in the form of a collection. It can’t. [↑](#footnote-ref-11)
12. In fact, STEP does have its own style sheet, which I came across once, but have never since managed to find again, and we were able to make a modification to this in order to fix a minor problem with rendering. But to respond to things like the fact that DBL specifies formats on a per text basis would presumably require STEP to accept a different version of its internal style sheet for each text, and there is, so far as I know, no mechanism to achieve this. [↑](#footnote-ref-12)
13. This was an issue to do with the way the text is rendered: it was not an artefact of the conversion process. [↑](#footnote-ref-13)