**STEPBible module converter**

Module converter



**User and maintenance Guide**

**Contents**

[1 Introduction 5](#_Toc179042560)

[1.1 The data formats 6](#_Toc179042561)

[1.2 OSIS … and OSIS 7](#_Toc179042562)

[1.3 Non-compliance 8](#_Toc179042563)

[1.4 Restructuring: Cross-boundary markup 9](#_Toc179042564)

[1.5 Reversification 10](#_Toc179042565)

[1.6 osis2mod 11](#_Toc179042566)

[2 Caveats 13](#_Toc179042567)

[2.1 Inputs 13](#_Toc179042568)

[2.2 Flexibility 13](#_Toc179042569)

[3 Using the converter – process overview 14](#_Toc179042570)

[4 Installing the tool and configuring the environment 15](#_Toc179042571)

[4.1 Before you start 15](#_Toc179042572)

[4.2 Program environment 15](#_Toc179042573)

[4.3 JAR file 15](#_Toc179042574)

[4.4 Folder structure 15](#_Toc179042575)

[4.5 Environment variable 15](#_Toc179042576)

[5 Creating a per-text folder structure 17](#_Toc179042577)

[5.1 The per-text root folder – location 17](#_Toc179042578)

[5.2 The per-text root folder – name 17](#_Toc179042579)

[5.3 The abbreviated Bible name 17](#_Toc179042580)

[5.4 The content of the per-text root folder 18](#_Toc179042581)

[6 Configuration and metadata 19](#_Toc179042582)

[6.1 Overview 19](#_Toc179042583)

[6.2 Configuration data – an outline 19](#_Toc179042584)

[6.3 Externally-supplied configuration data 20](#_Toc179042585)

[6.4 Obsoletes 21](#_Toc179042586)

[7 Do you need to pre-process the text? 22](#_Toc179042587)

[7.1 DIY 22](#_Toc179042588)

[7.2 Regex processing 22](#_Toc179042589)

[7.3 VL-specific pre-processing 23](#_Toc179042590)

[7.4 Applicable to XML-based formats 23](#_Toc179042591)

[8 Running the converter from the command line 25](#_Toc179042592)

[8.1 The command line 25](#_Toc179042593)

[8.2 Command-line options 25](#_Toc179042594)

[8.3 What a run looks like 28](#_Toc179042595)

[8.4 Outputs 28](#_Toc179042596)

[8.5 Parallel processing, progress- and error- reporting 29](#_Toc179042597)

[8.6 Additional information: the *TextFeatures* folder and the enhanced Sword configuration file 29](#_Toc179042598)

[8.7 Shared metadata and the repository package 30](#_Toc179042599)

[9 Philosophy and implementation 32](#_Toc179042600)

[9.1 Overview 32](#_Toc179042601)

[9.2 Code structure – background 33](#_Toc179042602)

[9.3 Code 33](#_Toc179042603)

[9.4 Parallel processing 33](#_Toc179042604)

[10 Tools etc 35](#_Toc179042605)

[11 OSIS conversion 36](#_Toc179042606)

[11.1 STEPBible OSIS 36](#_Toc179042607)

[11.2 A special note on cross-references 37](#_Toc179042608)

[12 A note on debugging 39](#_Toc179042609)

[13 Gotchas and arcane information 40](#_Toc179042610)

[14 Intellij IDEA 42](#_Toc179042611)

[14.1 Run configurations 42](#_Toc179042612)

[14.2 Creating JAR files 42](#_Toc179042613)

[14.3 Bugs 43](#_Toc179042614)

[15 OSIS: Detail 45](#_Toc179042615)

[15.1 General things to be aware of 45](#_Toc179042616)

[15.1.1 Span-type tags and verse markers 45](#_Toc179042617)

[15.2 Arbitrary tweaks with no significant implications for compliance or semantics 45](#_Toc179042618)

[15.2.1 Verse-ends 45](#_Toc179042619)

[15.2.2 Canonical headings 45](#_Toc179042620)

[15.2.3 Speaker tags 46](#_Toc179042621)

[15.2.4 Acrostic tags 46](#_Toc179042622)

[15.2.5 Selah tags 46](#_Toc179042623)

[15.2.6 Avoiding suppression of similar verses 46](#_Toc179042624)

[15.2.7 Notes – rendering of callouts 46](#_Toc179042625)

[15.2.8 Loss of poetry lines 47](#_Toc179042626)

[15.2.9 Blank lines 47](#_Toc179042627)

[15.3 Changes which may alter the structure or appearance of the text 47](#_Toc179042628)

[15.3.1 Missing verses 47](#_Toc179042629)

[15.3.2 Elisions 47](#_Toc179042630)

[15.3.3 Tables 48](#_Toc179042631)

[15.3.4 Plain vanilla paragraphs 48](#_Toc179042632)

[15.3.5 Indented paragraphs 48](#_Toc179042633)

[15.3.6 Introductory material 48](#_Toc179042634)

[15.4 Non-compliance 48](#_Toc179042635)

[15.4.1 Enclosing tags [NON-COMPLIANCE] 48](#_Toc179042636)

[15.4.2 Copyright information [NON-COMPLIANCE] 48](#_Toc179042637)

[15.5 Loss of semantics 49](#_Toc179042638)

[15.5.1 Speaker tags [LOSS OF SEMANTICS] 49](#_Toc179042639)

[15.5.2 Acrostic tags [LOSS OF SEMANTICS] 49](#_Toc179042640)

[15.5.3 Selah tags [LOSS OF SEMANTICS] 49](#_Toc179042641)

[15.5.4 Canonical titles at the ends of psalms [LOSS OF SEMANTICS] 49](#_Toc179042642)

[15.5.5 Notes and cross-references functionality [LOSS OF SEMANTICS] 49](#_Toc179042643)

[16 Tables 50](#_Toc179042644)

[17 Canonical titles at the start of Psalms [POSSIBLE LOSS OF SEMANTICS] 51](#_Toc179042645)

[17.1 General points 51](#_Toc179042646)

[17.1.1 Replacing psalm:title [LOSS OF SEMANTICS] 51](#_Toc179042647)

[17.1.2 Reversification footnotes 51](#_Toc179042648)

[17.2 Using the Crosswire mode of osis2mod with one of their schemes 52](#_Toc179042649)

[17.2.1 Ps 3 52](#_Toc179042650)

[17.2.2 Ps 11 52](#_Toc179042651)

[17.2.3 Ps 51 52](#_Toc179042652)

[18 Useful places to check output 55](#_Toc179042653)

# Introduction

This document talks about the STEPBible Text Converter. It tells you how to use it, and it gives various miscellaneous information which may be helpful if you are landed with the task of maintaining it.

Here’s a useful schematic showing what it does, which will help you distinguish it from other lesser forms of software:

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

In other words, it takes a Bible text in USX, VerseLine (VL) or OSIS form, and converts it to a Sword module, which it zips up, along with other information, in a package for storage in the STEPBible repository. These formats are discussed briefly in section 1.1. (I’ve also worked with Crosswire IMP format, but support for that is rather ad hoc.)

|  |
| --- |
| **If your main aim is merely to *use* the converter, feel free to skip now to section 3. If you need to understand or maintain it, read on for an overview of the internals.** |

## The data formats

|  |
| --- |
| **USX** is an XML dialect maintained by UBS, and is 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. |
| **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 by way of input; second, we may need to apply manual tweaks to some texts (eg to add tagging), and OSIS is regarded as the best basis for 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. |
| **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), or how these additional features should be represented. 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 have made this configurable 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. |
| **IMP** (which I mentioned in passing previously) is a Crosswire-proprietary text-based format which can be extracted from Sword modules using the Crosswire *mod2imp* utility. We have successfully worked with this a few times, but it is not really officially supported – a lot of manual tweaking may be involved, because it (so far as I know) undocumented, and we therefore need to change the processing to accommodate things as we find them. It is also worth noting that Crosswire do not recommend reliance upon *mod2imp*. |

## OSIS … and OSIS

It may help make sense of later discussions to know that processing uses / creates two different OSIS files.

There are essentially three different ways in which processing may proceed:

* The text may have been supplied to us in non-OSIS form (USX, VL, etc), and we may be starting from that. Here the converter begins by creating an initial OSIS representation of the data, applying as little processing as possible, so that the result is as close as possible to the original. This OSIS representation I refer to here as *external-facing* OSIS. It is saved on disk in case we need it in future.
* We may have only OSIS as a starting point. In this case, this OSIS *is* (more or less) the external-facing OSIS.
* We may have started from a non-OSIS input previously, which will have generated external-facing OSIS; but we may now have made manual changes to that OSIS (for example to apply tagging), and may want to create a module directly from that. In this case, this modified OSIS continues to function as the external-facing OSIS.

This external-facing OSIS is important precisely because it *is* the form to which we may apply tagging etc. (It is also suitable to pass to third parties should we wish to do so, although there are some caveats here, because the OSIS we generate isn’t quite compatible with the standard – see section 1.3.)

And then there is a second form of OSIS, which I generate from the external-facing OSIS, and which I call, with stunning originality, *internal-facing* OSIS. This is really private to the converter, and I’d rather no one knew about it, but since I may leave it lying around for debugging purposes, I guess I’ll have to come clean.

So the reason for its existence is that it is desirable to apply quite a lot of additional processing to the data in order to work around shortcomings in the STEPBible rendering, to perform validation, and to avoid constructs which would get in the way of our other activities (such as reversification). The fact that there is so much of this processing – along with the fact that in some cases it is rather ad hoc – means that its output is not necessarily all that stable: I may need to alter the processing quite frequently to accommodate new texts, and as a result the details of the internal-facing OSIS may change.

A form of OSIS which is likely to change quite frequently does not seem a particularly good candidate for us to use as a basis for additional modifications such as tagging, because if that activity is automated at all, it may rely upon its input *not* changing too much. I therefore retain the external-facing OSIS for that purpose, and then create from it this throw-away internal-facing OSIS for the purpose of actually generating the module.

## Non-compliance

I mentioned above that the external-facing OSIS is not quite compliant with the OSIS standard.

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 (and indeed with nested lists, difficult to know what they are supposed to look like); adding them increases the likelihood of hitting cross-boundary markup (see section 1.4); if we have them, they introduce excessive vertical whitespace into the rendered text; and things seem to work perfectly well (for us, at least) 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 OSIS 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 provide third parties with our Sword configuration file. Some of the items contained in that file can be quite extensive (for example, the ‘About’ field which describes a text), and when they show up on STEPBible’s copyright page, they are much easier to read if they are in HTML form.

We therefore use HTML for quite a number of the configuration parameters.[[2]](#footnote-2) Strictly, we should not do this. Crosswire officially accept HTML in only a very few fields, and even there it is a very limited subset of HTML. In other fields, we are supposed to limit ourselves to using Crosswire’s very restricted markdown language. That, however, is rather restrictive, and since it seems a lot of people ignore these limitations, we do so too.

## Restructuring: 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 …

* With a milestone marker at the front of the given entity only.[[3]](#footnote-3)
* Or (in some versions) with a milestone marker at front and end.
* Or with an enclosing marker.

OSIS, as I recall, requires that books be enclosing nodes, and allows chapters and verses to be either milestones or enclosing tags, but recommends the former, at least for verses.[[4]](#footnote-4)

Books and chapters can easily be swapped from one arrangement to the other because there is normally no cross-boundary markup[[5]](#footnote-5). But the milestone form of markup for *verses* is more problematical, because it permits, and by implication encourages, semantic and formatting markup to cross verse boundaries.

This is fine for printed works, which presumably is what it is largely aimed at. But it is very awkward for electronic texts where there may be a requirement to access verses individually out of context, because cross-boundary markup makes it 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. Clearly the verse-end for verse *n* must come before the verse-start for verse *n* + 1, but we do have the liberty to position the verse-end anywhere between the two verse starts, provided only that no canonical text falls outside the verse, and it is often useful to take advantage of this fact.

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 an empty self-closing paragraph marker at the front of the original paragraph, and do the same with poetry paragraphs[[6]](#footnote-6); 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 at all). 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, rather than *osis2mod*, 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 STEPBible’s added value features. For instance, the vocabulary STEPBible displays for Dan 1:2 is based upon the mouse hovering over a verse marker which reads ‘Dan 1:2’ and then looking up the vocabulary for Dan 1:2. If Dan 1:2 in different Bibles reflected different portions of the underlying ancient text, the vocabulary would – without further work – be wrong for some of them.

At one time we supported two different ways to address this.

One, which I referred to as *conversion-time* reversification, entailed restructuring the text during the conversion process – moving verses around and relabelling them, so that the generated module was 100% NRSV(A) compliant (NRSV(A) being our chosen standard). In general, licence conditions preclude quite such major surgery, so we never anticipated making much use of this; and at the time of writing we are rather assuming that in fact we will *never* use it. I have retained at least some of the code in the converter, but it is not used – and indeed probably would now require some fairly significant changes before it could be made to work.

The alternative – *runtime* conversion – leaves the text as-is through the conversion process. This almost certainly means the text conforms to none of the schemes built into the Crosswire *osis2mod* utility. In turn, this requires that we build the module using our own version of *osis2mod* which can handle texts in this form, and that it be rendered using our own version of JSword.

Modules built using runtime conversion are displayed using exactly the same versification as specified by the translators. This is a major advantage in that it means we avoid the kind of restructuring which might be precluded by licensing conditions, and also in that users who are familiar with the structure of the original text are not confronted with something which mysteriously deviates from it.

When using added value features within STEPBible, deviations from our standard NRSV(A) versification scheme are automatically taken into account on the fly.

Strictly, there is also a third reversification option – that of not reversifying the text at all. This option *must* be used when creating public modules, because those *have* to use Crosswire’s *osis2mod*.

## 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 STEPBible form.

For present purposes, ignore the Crosswire version. Our own version can function in either of two modes – Crosswire-compatible or STEP-specific – depending upon the arguments supplied to it.

We use Crosswire-compatible mode when creating public modules because these need to work with standard libraries like JSword etc. These do have a downside, however: they have to fit one of the versification schemes supported by Crosswire. The first issue is that almost no Bible ever does fit any scheme exactly, and the Crosswire processing may therefore restructure the text somewhat. And the second issue is that if a Bible is not NRSVA-compliant, STEPBible’s added value features won’t work with it correctly.

STEPBible-specific mode gets round these issues – the versification scheme actually used by the Bible is respected, but additional processing ensures NRSVA-compliance when using the added value features. This, however, also comes at a cost: the resulting module will work only with our own versions of things like JSword.

STEPBible-specific mode *must* be used if we are using run-time reversification or if any verses or books are out of order.

**User guide**

# Caveats

## Inputs

USX and OSIS are complicated standards. No two translators seem to agree fully on how to use them. In addition, for all their complexity, they also have limitations, and translators attempt to work around these, each in their own way. This means we often have to cater for situations we have not encountered before. As a result, the converter may give up on new texts more often than you might want (probably a truism, since you’d rather it didn’t give up at all, presumably …)

Similarly, there are issues with VL, because VL is not actually a standard, and we therefore have no way at all of anticipating what new texts will look like. I’ve tried to make some provision for this by making aspects of the processing configurable, but there is no guarantee that it is configurable *enough*.

Note also that it is only relatively recently that we have been taking OSIS and IMP as inputs, and so we have not exercised this aspect of processing particularly thoroughly at the time of writing (August 2024).

## Flexibility

The converter has been designed to be very flexible. This, of course, is because I don’t know what I’m doing. On that basis, flexibility is useful. On the other hand, explaining that you could do things this way, or that way, or possibly even the other way, makes for a very complicated user guide.

I have therefore taken the arbitrary decision that what works for me will also work for you, and that’s the only option I actually describe here. If by any mischance it *doesn’t* work for you, get back to me. There are almost certainly alternatives.

|  |
| --- |
| Using the converter – process overview  1. Install the converter and configure its environment. See section 4. 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 and outputs for your text. See section 5. 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 5. 3. Set up configuration information. See section 6. 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 7. 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 8. |

# Installing the tool and configuring the environment

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

## Before you start

Hopefully you should find that (with suitable changes to configurable file paths) the converter works on both Windows and Linux, and that the following instructions apply to both. Note that you should use Linux format for file paths (ie with slashes rather than backslashes as separators within paths) regardless of which platform you use.

## Program environment

* You need to have Java 18 or later installed.
* You need the STEPBible version of the *osis2mod* program. (To state the blindingly obvious, you will need versions which run on the platform upon which you are working – Windows, Linux, etc.)

## JAR file

Store the TextConverter.jar file somewhere convenient.

## Folder structure

Create the following folder structure:

StepTexts  
|  
+-- \_DebugOutput\_  
|  
+-- \_SharedConfig\_

In fact, it doesn’t matter what you call the root folder; I’m going to assume *StepTexts* in what follows, so I haven’t got to keep reminding you that you can do something different if you wish. And it doesn’t matter where you locate the *StepTexts* folder.

Under \_*SharedConfig*\_, store the initial configuration data you should receive as part of the installation package (if you receive it in zipped form, unzip it).

In due course, you will create individual folders for each text you work with. These will go – directly or indirectly – under *StepTexts*.

## Environment variable

Create an environment variable called *StepTextConverterParameters.*

The content of this should be of the form:

nameA=valueA; nameB=valueB; …

It should contain at least the following settings (you can add more if you wish):

|  |
| --- |
| **stepTextConverterOverallDataRoot=<path for StepTexts>**  eg **stepTextConverterOverallDataRoot=~/StepTexts** on Linux  **stepTextConvterDataRoot=C:/Users/Jamie/Documents/StepText** on Windows.  These *are* only examples – as I said above, you can store the folder anywhere you like. |
| **stepOsis2modFilePath=somePathOrOther**  The path to the STEPBible version of *osis2mod*. |

# Creating a per-text folder structure

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

## The per-text root folder – location

You need to create a separate root folder for each text.

These should go under *StepTexts* – either directly or under some substructure of your own devising. You may, for instance, find it convenient to group together all texts having some common provenance in some subfolder – all texts supplied from DBL, for example.

## The per-text root folder – name

|  |
| --- |
| ***All per-text root folder names should look like one or other of:***  ***Text\_eng\_XYZ\_xxx***  ***Text\_eng\_XYZ\_th\_xxx*** |

All names should start with *Text*, followed by additional portions separated by underscores as shown.

The **red** portion is the 3-character ISO language code in lower case.[[7]](#footnote-7),[[8]](#footnote-8)

The **green** portion is the abbreviated name of the Bible (eg KJV), following whatever capitalisation is normal for the text.

The **purple** portion is an optional suffix (often *th*, which is why I’ve shown it as this above). A suffix was added occasionally in the past to show that we (or Tyndale House) had made significant changes to the text. Where this was the case, we normally retain the suffix. We never add this suffix on new texts, however.

The **blue** portion indicates how this text should be handled. *step* indicates that we will generate an STEPBible module only from the text. *public* indicates we will generate only a publicly available text. *stepPublic* (or *publicStep*) indicates that we will generate both. This portion is not case-sensitive. If omitted, *step* is assumed. This is mainly for backward compatibility. On new texts, I recommend always including this portion for the sake of clarity.

## The abbreviated Bible name

The abbreviated Bible name (the green portion mentioned in the previous section) is complicated.

Very often the translators will have provided an abbreviated name for you. Where this is the case, if you have both an English and a vernacular form, use the vernacular latter if available and 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.

However …

* In some cases, the abbreviated name as supplied is simply a copy of the language code. Where this is the case, we tend not to use the supplied abbreviation. SeedCompany and UnlimitedBible texts often do this, for instance. For SeedCompany texts, we use ‘SC’ as the abbreviation. For UnlimitedBible, we use ‘ULB’.[[9]](#footnote-9) We will have to cater for other examples as we come across them.
* For Biblica open access texts, if the text has a copyright version which we already support, we use the abbreviation from that. (Typically the open access texts have names and abbreviations which add the word ‘open’ to the name, or a single-letter representation of ‘open’ to the abbreviation. Our aim is to drop that.)
* We don’t include in either the name or the abbreviation anything which might indicate that this is a partial text, on the grounds that more of the text may be translated in the future, and we don’t want to have to change the name.
* We try to align names with common forms available from the internet.

## The content of the per-text root folder

A typical per-text root folder needs to look like this before any processing occurs:

Text\_eng\_KJVA\_step  
|  
+-- InputUsx (or whatever)  
|  
+-- Metadata

In the *Metadata* folder, you need, at the very least, a file called step.conf, containing configuration information. We look at configuration information in more detail in section 6.

And you then need a folder to hold the input data. You can use any names for the input files.

|  |  |
| --- | --- |
| **USX:** | The folder should be called *InputUsx*. There should be one file per Bible book, and each file should have an extension of .*usx*. |
| **VL:** | The folder should be called *InputVl*. There should be a single file, with an extension of .*txt*. |
| **IMP:** | The folder should be called *InputImp*. There should be a single file, with an extension of .*imp*. |
| **OSIS:** | The folder should be called *InputOsis*. There should be a single file, with an extension of .*xml.* |

Don’t put any other files into the input folders or it may confuse the processing. If you need to co-locate other things with the input files, create a subfolder within the input folder, and put things there.

# 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 eg 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. Fortunately there are sensible defaults for most of the parameters which the system relies upon, so in general you don’t need to create huge amounts of configuration data for each new text. Plus the system does contain processing to pick up configuration data from DBL’s metadata.xml and license.xml files where you are working with DBL-originated material (provided you trust what they provide: we don’t always, but I give more details in section 6.3).

## Configuration data – an outline

You must create a file called *step.conf* and store it in the Metadata folder for the text.

This file will normally contain certain definitions of its own, and point out to other files for common definitions or defaults. If such a file already exists for a text similar to the one you are working with[[10]](#footnote-10), probably the easiest approach is to take the *step.conf* from that text and modify it.

You may find the command-line parameter –dbgConfigData generateStepConfig[All] useful here – it will generate a template file for you. See the discussion in section 8.2.

A detailed discussion of the way configuration data works appears in *\_READ\_ME\_.txt* in the *Resources* section of the converter JAR file[[11]](#footnote-11), so I won’t duplicate that in full detail here. It may help to have an outline of what’s involved, though:

* Configuration data comes from the command line used to invoke the converter, from the environment variable *StepTextConverterParameters*, from configuration files which you set up yourself, and from the *Resources* section of the JAR file (this last containing all of the default settings).
* Configuration files may contain directives which include other configuration files.
* The main element within a configuration file is the ‘definition directive’, which looks something like stepPreferredIceCream=Pistachio. This associates the value *Pistachio* with the name *stepPreferredIceCream*.[[12]](#footnote-12)
* Definitions can also make use of other definitions, so that, for instance, you can build a complex value out of a number of simpler ones.
* It is possible to have multiple definitions for the same item. The configuration processing defines rules for priority in such cases

Just to give a feel for what a *step.conf* might look like, a simple example appears below. Do not take this as a basis of any of your own work, however, because the configuration requirements change from time to time, and I probably won’t remember to keep the details here up to date.

Lines starting #! are comment lines. The History information at the end of the file is kept up to date automatically by the converter – you won’t normally change it yourself.

This example is a particularly simple file because the text to which it applies comes from Biblica, and therefore a lot of the configuration information is common to all Biblica texts, and is included from a common file via the $include statement. The *@(…)* portion of this, incidentally, is an example of the facility mentioned above, whereby a value previously associated with a configuration parameter can be used later within the configuration processing: *stepTargetAudience* will have the value *public* or *step*, and the processing will therefore include one of two files, as appropriate to the target audience.

|  |
| --- |
| #!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  stepBibleNameVernacular#=মুক্তভাবে বাংলা সমকালীন সংস্করণের  stepVersificationScheme=NRSV  $include $find/Biblica/biblicaCommonConfig\_@(stepTargetAudience).conf  #!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  History\_1.0=2024-08-12 [SupplierVersion: DBL:7] SupplierReason: Remove checkboxes; StepReason: OpenAccess first release. |

## Externally-supplied configuration data

In some cases, we may be given configuration data by the text supplier in a form which is relatively easy to process. In such cases, it may be possible to include within the converter processing which will automatically extract the data from that file, so removing the need to transcribe it.

At the time of writing, we support this only for texts supplied by DBL, where we are able to take data from their metadata.xml and license.xml files (the latter purely for administrative purposes).

*\_READ\_ME\_.txt* in the *Resources* area of the JAR explains how to take advantage of this (or points to where you can find the information).

Note, though, that latterly we have tended to be rather less prepared simply to accept this data as-is.

## Obsoletes

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.

# Do you need to pre-process the text?

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

***described here is run automatically every time the converter runs.***

The input formats we cater for are complicated, and people get them wrong. Even if they get them right, the data 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 each text is likely to have its own unique issues.

There are a number of options for addressing the shortfall. (You may not become aware of the need for these, though, until you have carried out an initial run without them and discovered that there are issues in the output.)

## DIY

You can, if you wish, simply modify the text yourself before you provide it to the converter. This gives the ultimate in flexibility. However, if you ever receive a revised version of the text, you need to remember to apply the processing again (assuming the revised version does not, itself, fix the issues).

The alternative is to rely upon one of the automated processes described below. These, too, come with drawbacks, though. Automated processing relies upon recognising relevant patterns in the data, and if you receive a revised version of the text, those patterns may not still apply.

## Regex processing

This can be applied to all forms of input.

You supply one or more lines starting *stepOsisRegex* and / or one or more starting *stepNonOsisRegex*.

An individual line looks like:

stepOsisRegex=A(.+)B => AB

The definition portion gives a regular expression, followed by ‘=>’ (optionally set off by spaces before and after), and then optionally a replacement value. (If the replacement value is absent, then matches are deleted.)

The left- and right- hand sides of the => can be anything acceptable as arguments to the Kotlin *replace* statement.

You can have multiple *stepOsisRegex* lines and or multiple *stepNonOsisRegex* lines. They are applied to the raw text in the order specified before they it is passed on for further processing.

Separating OSIS and non-OSIS definitions gives the opportunity to apply both during a given processing run, although in most cases I’d recommend *either* the one *or* the other.

You can, if you wish, use regex pre-processing in combination with any of the other forms of pre-processing.

## VL-specific pre-processing

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.

## Applicable to XML-based formats

At the time of writing, this is applicable to USX and OSIS data.

You can supply fragments of XSLT via configuration parameters *stepXsltStylesheetForGeneralInputPreprocessing* and *stepXsltStylesheetForForcedOsisPreprocessing*.

Recall that you can start a run from whatever input you happen to have available, or (where you have done a previous run which created OSIS and perhaps have modified that OSIS), you can force the OSIS to be used as input instead.

If you have forced the issue, *stepXsltStylesheetForForcedOsisPreprocessing* is applied. Otherwise, *stepXsltStylesheetForGeneralInputPreprocessing* is applied.

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:[[13]](#footnote-13)

stepXsltStylesheet#= \

<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>

You can, if you wish, use XSLT pre-processing in combination with any of the other forms of pre-processing.

# 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 several entirely separate purposes.

The main one, naturally, is to generate a module and its containing repository package from the various inputs. But you can also use to perform various checks and / or to generate various kinds of supporting information. These uses are discussed in subsequent sections.

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.

## The command line

The converter is run from the command line as below:

java -jar PathToJarFile\TextConverter.jar [options]

## Command-line options

Some options can only be applied from the command line. Others can alternatively be supplied from within configuration files.

All of these are converted, for the benefit of the processing, to configuration parameters. As such, they have step pre-prepended to their names. Thus *rootFolder* becomes *stepRootFolder*. Bear this in mind if you opt to store any of the data below in configuration files rather than supplying it via the command line – you will need to include the initial ‘*step*’ yourself.

Parameters in pink below are mandatory.

|  |  |
| --- | --- |
| -rootFolder <path> | **Mandatory -- on command line.** The root folder for the text being processed. This must be supplied on the command line. |
| -releaseType <type> | **Mandatory – command line or config file.** Options are Major or Minor, indicating the type of release covered by this module. A major release gives rise to a full increment in the version number. A minor release gives rise to a dot increment. Typically we’d regard a release based upon new material from the supplier as major (although there may be circumstances where minor is more appropriate). Where the release reflects a change we’ve made ourselves, we need to consider how significant that change was. |
| -supplierUpdateReason <text> -stepUpdateReason <text> | **Mandatory – command line or config file (probably best on command line).** The reason the supplier provided revised text, or, where we have changed the module ourselves, the reason for doing so.You must supply at least one of these, and may supply both. They appear on the History line to explain why the module was generated. |
| -conversionTimeReversification | **Not supported at present – we no longer think this will be used. I have retained some of the handles needed to access the relevant code, but at present it is almost certainly not even working.**  **Optional – probably best in a config file.** If we produce a public-facing module, we do not apply reversification. If we produce a STEPBible-internal one, normally we use runtime reversification. For a few modules (modules aimed mainly at an academic audience and not subject to restrictive licence conditions), we may opt to physically restructure the module during the conversion process. This flag lets us force that.  **\*\*\* Note that this option has not been exercised for a long time. \*\*\*** |
| -targetAudience <type> | **Optional – probably best on the command line.** Ifwe are processing a text under a root folder (*Text\_...\_step*) ends in *step*, we know that the target audience is STEPBible. If it ends in *public*, we know we are building a public module. If it has no distinguishing suffix, it is as though it ended with *step*. If it ends with *stepPublic* or *publicStep*, however, we could be building either, and in this case this parameter is needed to indicate which (‘type’ is either *step* or *public*). Two runs will be needed for such modules, one for each setting. |
| -forceUpIssue | **Optional – probably best on command line.** I’m not sure about the wisdom *of* this one … Past experience is that we normally need to carry out several runs on a given text before we get it right. When we did that, the converter was up-issuing things each time, which was a pain, because the intermediate attempts were meaningless. I therefore changed the processing so that if the ‘reason’ details didn’t change from one run to the next, I didn’t up-issue. But this then meant I needed some way to force an up-issue where the reason didn’t change but I still needed a new version number. |
| -manualOsis2mod | **Optional – command line or config file.** I always attempt to run *osis2mod* directly under control of the converter, so that the entire job can be completed without further intervention. Occasionally, however, *osis2mod* seems to stall under such circumstances. If you supply this parameter, the processing will copy to the clipboard the command you need so you can run *osis2mod* manually, and will then wait until you inform it *osis2mod* has completed. |
| -useExistingOsis | **Optional – command line or config file.** Most texts are supplied to us as USX, VL, etc. Where this is the case, the converter generates OSIS during the conversion process and saves it. Sometimes we may wish to take previous OSIS as the starting point and ignore the existence of the USX or whatever (for example when we have applied tagging to the OSIS). Options are *AsInput* (use the existing OSIS, but apply to it any changes deemed necessary) or *AsOutput* (use the existing OSIS as-is). The latter option was introduced for ESV, where we believed what we had was already correct, and didn’t want to risk it being changed. |
| -help | **Optional – command line.** Outputs help information. (Help information is also output if you run the converter with no arguments or with invalid arguments.) The run does not generate a module, and does not affect any existing data. |
| -version | **Optional – command line.** Outputs the version number of the converter JAR file. The run does not generate a module, and does not affect any existing data. |
| -checkInputsAgainstPreviousModule | **Optional – command line.** Checks the SHA256 digests for the inputs against the digests used on the most recent previous build of this module and reports any differences. The run does not generate a module, and does not affect any existing data. |
| -evaluateSchemesOnly | **Optional – command line.** Compares the versification structure of the text with the various versification schemes supported by Crosswire and gives a goodness-of-fit score for each. Useful where you intend to generate a public-facing module, because these have to be processed using the Crosswire mode of *osis2mod*, and we therefore have to select a versification scheme for them. The run does not generate a module, and does not affect any existing data. |
| -dbgConfigData <details> | **Optional – command line.** Options for ‘details’ are some combination of *reportSet* (reports on stdout details of which configuration parameters are set and how they are set); *reportMissingDebugInfo* (reports details of configuration parameters for which I have no descriptive information in *Resources/configDataDescriptors.tsv*); *generateStepConfig* or *generateStepConfigAll* (generates on stdout a template *step.conf* file: with ‘All’ that file contains pretty much all of the parameters you might ever normally want to change; without ‘All’ it contains only the more common ones. |
| -dbgAddDebugAttributesToNodes | **Optional – command line or config file.** Adds attributes to nodes as they are processed, giving more information about what has been going on. Haven’t used this for a long time, so I don’t know how useful it is, nor even if it actually works. |
| -dbgDisplayReversificationRows  <type> | **Optional – command line or config file.** Displays those reversification rows which have been selected. ‘type’ may be anything containing ‘screen’ to send output to stdout, or anything containing ‘file’ to send output to debugLog.txt. You may also give both to send output to both locations. Include ‘deferred’ if you want screen output at the end of the run, rather than as it occurs (where it might be interleaved with other output). |
| -dbgSelectBooks <abbrevList> | **Optional – command line or config file.** Lets you limit processing to selected books only (a comma-separated list of abbreviations). Intended mainly to speed things up when debugging. I think it works ok with USX as input; not sure about other things. |

## What a run looks like

On very large texts, conversion may take a number of minutes. To keep you amused, the converter writes a *lot* of progress information to stdout (useful also when it goes wrong, as it will).

Assuming things work, this output may end with a couple of messages in very large text indicating anything deemed particularly important about the run. For example, you are informed if encryption was not used. Big text is usually good, not bad – unless it indicates that the processing has done something you didn’t anticipate.

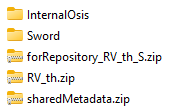
Each run of the converter generates two log files – *converterLog.txt* and *osis2ModLog.txt –* in the per-text root folder. The latter contains messages generated by the *osis2mod* program, and the former contains messages reflecting the overall conversion process. *converterLog.txt* contains copies 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.

## Outputs

A single successful run generates a single version of the module.

If the run was for a STEPBible module, the run will give rise to a folder called \_ *Output\_Step* directly under the root folder for the text. If it was for a public module, it will give rise to \_ *Output\_Public*. Where a single text is permitted to generate both a STEPBible and a public module, you will need to run the converter twice, and will end up with each of these two folders.

The content of these output folders typically looks something like:



The most important parts of this are:

* The *forRepository* file, which contains various information to be stored in the STEPBible repository – a copy of the module itself, along with other information which may be needed to rebuild the module in future, and various information describing the module which may be useful for admin purposes or for locating modules with particular characteristics in case we need to rebuild them.
* The second zip file (RV\_th.zip above), which is the actual module. Strictly, this is not that valuable, because a copy also appears in the repository package, but it is convenient to have it separate while trying things out.
* The *InternalOsis* folder may be useful for debugging purposes – it contains the OSIS which was actually used to generate the module.
* A copy of *sharedMetadata.zip* ends up in the repository package, from which point of view there is really no need to retain *sharedMetadata.zip* itself. I have to admit I’m not entirely sure about the need for this.
* Strictly speaking, the other items – the *Sword* folder and *sharedMetadata.zip* – could be deleted: they aren’t needed again and are only taking up space.

## Parallel processing, progress- and error- reporting

Progress and errors are reported to stdout and stderr. By default, the converter attempts to do as much work as possible in parallel in order to speed things up.

One downside to this is that console output may become slightly muddled in a few places. In particular, when running over multiple books, there is no guarantee that the books will be handled in any particular order – neither alphabetical order nor Bible order. If this turns out to be a problem you can revert to sequential processing, rather than parallel processing, by setting *stepPermitParallelRunning* to No in a configuration file or in the *StepTextConverterParameters* environment variable, or by giving it as  
-*PermitParallelRunning No* on the command line. In particular, it will probably be useful to turn off parallel processing when debugging (always assuming, of course, that what you need to debug is not an issue introduced by parallel processing …).

More information about parallel processing appears in section 9.4.

## 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. These are copied to the repository package.

*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.

## Shared metadata and the repository package

I mentioned above that the repository package contains a copy of all of the shared metadata upon which a module may have relied.

I have to admit I’m in two minds about the desirability of this.

On the one hand, it does seem useful to have available all of the inputs which fed into a module. On the other hand, one important thing that we don’t have available in the repository package is the converter JAR itself, and the latest version of the JAR file may now have different requirements as regards metadata (although from now on I will do my best to ensure changes are backwards compatible). Plus there is always the possibility that we may have applied corrections to the shared metadata now being used when generating new modules, and we may want these changes to be applied if we regenerate the module.

So, working with the old metadata … desirable or undesirable; feasible or infeasible?

**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, which are, in turn, gathered together with other data to create a package of information for the STEPBible repository.

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). And also, in a somewhat half-hearted manner, IMP; I haven’t shown this below because support probably isn’t all that good.

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 |

Unfortunately, this probably isn’t all that enlightening, and indeed may be somewhat confusing. So a little more explanation …

It is convenient to split the world into two – texts for which all we have by way of input is OSIS, and texts where we have something else available.

\*

Taking the OSIS-only texts first, the OSIS available to us will originally have come from some third-party source. Since then we may or may not have modified it ourselves outside the ambit of the converter (for example to add tagging).

I retain this form of the OSIS for possible future use (it serves as what I called *external-facing OSIS* in the introduction). The converter then modifies it in-memory in various rather ad-hoc ways to create something I refer to as *internal-facing OSIS*. This it stores separately and uses as input to *osis2mod*, following which it generates the module and the repository package. In theory the internal-facing OSIS is of no long-term value and should be thrown away after use, although in fact I retain it in case I need to refer to it if problems are identified.

\*

Moving on to the texts for which we have something other than OSIS available …

On the first run for a text, the converter creates OSIS from these inputs, applying as little processing as possible. It then stores this OSIS, to serve in future as external-facing OSIS. After this, the remainder of the process is as above.

The real difference comes in that the external-facing OSIS created on the initial run can now optionally be used, for example, for tagging. This means that on future runs, I have to know whether to start from the OSIS (in which case processing is as for OSIS-only texts), or whether to start from the non-OSIS inputs (in which case processing is as described in the previous paragraph). Some care is needed, because if tagging is applied but we then later start with the non-OSIS inputs, the tagged OSIS will be overwritten.

## Code structure – background

A note on the history of the converter. Previous versions of the converter assumed USX as a starting point – if we had USX, that was what we used, and if we had VL or IMP, we converted that to USX and then proceeded from there. This meant that it was reasonable to apply all of the complex processing of the conversion activity to USX.

However, we now have to cater for the possibility that we will start from OSIS on some runs. Given this requirement – and given the fact that *all* runs generate OSIS en route to creating a module, it made sense to alter the processing so it was based upon OSIS rather than USX. At the same time I didn’t quite have the courage of my convictions, so I have tried to write as much as possible of the revised code in such a way that it will work upon either OSIS or USX (sadly, of course, thereby increasing the complexity of the code somewhat). Bear in mind, though, that at present we have no reason to exercise it on USX, so I cannot guarantee that it will work.

## Code

The code is heavily commented (and the comments may even – eventually – be up to date). Rather than repeat everything here, I refer you to the generated HTML documentation set.

Each package has (or should eventually have) an interface called AAA\_Doc which contains general information about the package.

## Parallel processing

As far as possible, I have organised the processing to be able to run in parallel, whilst at the same time making it very easy to turn off parallel processing with only minimal effort.

You will find numerous instance of *with(ParallelRunning(true))* in the code. If you change the *true* to *false*, that particular portion of parallel processing will be turned off, but others will continue to operate. Alternatively, you can use a configuration parameter to disable parallel processing altogether – see section 8.5 for details.

There are really two ways in which it would be possible to take advantage of parallel processing: I could have one thread for each book to be processed, and then run the various steps for that book sequentially on that thread; or I could run the steps themselves sequentially at the top level, and within each step could process the individual books in parallel. I have opted for the latter on the grounds that I think it makes it easier to see what’s going on. So, for example, at one point I process elisions on all books in parallel, and at another I process tables on all books in parallel.

I currently apply parallel processing on USX input (where typically each book comes in a separate file, making the split easy); and on the processing which generates the internal OSIS (where I arrange for each book to end up temporarily in its own DOM).

Parallel processing does have downsides:

* Screen output may come out in the wrong order and / or interleaved. In fact there are few places where it is actually interleaved, but certainly where books are being processed in parallel there is no guarantee that they will be announced in any recognisable order.
* If one of the threads throws an exception, I cannot actually force all threads to terminate immediately: I can *request* that they do so, but they complete in their own sweet time. As a consequence it is perfectly possible that further error messages may be output. I *think* I manage to output information relevant to the first such error, but I have no way of being sure.

Rather late in the day, it occurred to me that initialising Kotlin objects in parallel might be a problem. There are a number of objects which contain what is essentially static data which needs to be initialised just once. Kotlin init blocks are not inherently thread-safe, and I did wonder what would happen if thread A required to make use of one of these objects, and therefore started initialising it, only for it then to be swapped out in favour of thread B, which also wanted access and set out carrying out its own initialisation.

To get round this, I thought it might be useful simply to make reference to all of the various objects as early as possible in the overall processing (somewhere before the start of parallel processing). My original intention was to use reflection to identify the objects and then instantiate all of them. That way adding or removing objects would not be a problem – the code would automatically find all of them.

This looked initially as thought it was going to be difficult – sample code on the web was complex, and there were suggestions that perhaps it was not particularly robust. I therefore created an interface – ObjectInterface – which did absolutely nothing, but from which all objects could inherit (it being rather simpler to use reflection to find things which inherited from a particular interface). The downside to this, of course, was that I needed to remember to add ObjectInterface, but in the belief that this was my best option, I set about doing this.

I then discovered a much simpler way of identifying all objects which did not rely upon this trick (and therefore did not rely upon me remembering to make things inherit in this way), and I now use that instead. However, by that point I’d already added ObjectInterface to all objects which I thought would benefit, and I have left it in place just in case it turns out to be useful at some point in the future.

**IMPORTANT**: It turns out that the DOM is not thread-safe, even when merely reading from it. Or in fact, possibly it is thread safe – it’s always possible that it’s the wrapper I put round the DOM accesses which was the problem, in which case I’ve fixed it.

# 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.
* **configDescriptions.xlsx**: Holds descriptions of all of the acceptable configuration parameters – both those which may be supplied by the user and those which may be generated internally. Used to generate a tab-separated variable file stored within the Resources section of the converter, which permits configuration accesses to be controlled and debugged.
* **isoLanguageCodes.xlsx**: Pretty much what it says on the tin. Used to generate a data file stored in the Resources section of the converter.
* **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

## STEPBible OSIS

The bulk of the most complex processing is devoted to turning external-facing OSIS into internal-facing OSIS. Why the two versions?

It takes a lot of complicated processing to produce a form of OSIS which we can pass to *osis2mod* and ultimately render correctly – some of this processing rather ad hoc in nature as I will explain shortly. A lot of complex processing is a lot of complex processing which is likely to contain bugs, or for which the requirements are likely to change a fair bit. And this in turn means that even if our inputs remain fixed, this version of the OSIS is quite likely to change relatively often.

It didn’t seem to me that this was a good platform to which to apply changes such as tagging: I presume that tagging is sufficiently time-consuming that you will want to automate the job, and that such automated processing may well assume that the general shape of the OSIS doesn’t change too often.

If this messed-around-with OSIS – what I have referred to elsewhere as *internal-facing OSIS* – isn’t up to the job, then I needed something else which could be used for tagging, etc.

Which is where *external-facing* OSIS comes in. The idea is simply that I take the original inputs and transform them into a valid OSIS representation with the absolute minimum of processing. This won’t be what we need to feed to *osis2mod*, but at least with minimal processing, we have less processing to change, and therefore it is less likely that we will make changes – resulting in a form of OSIS which is likely to prove relatively stable.

Some of the activities involved in generating the internal-facing OSIS do have what might be seen as a real-world motivation – things like adding annotation where verses were omitted in the original, or applying reversification processing. More specifically (although still only in outline), in getting to the internal-facing OSIS, we do the following:

* *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 and we’d prefer to work with something more uniform.
* *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, retagging material to avoid cross-boundary markup, or changing the verse structure of the text.[[14]](#footnote-14)
* *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.)

As regards circumventing bugs, the following list is probably not exhaustive, but will at least give an idea of what is involved:

* I fill in missing verses because *osis2mod* / *JSword* go wrong if verses are absent.
* I replace some semantic tags with formatting tags because STEPBible does not render the semantic tags in question in an acceptable manner.
* 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.
* 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.
* 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.
* 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.
* 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;).
* 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.

## A special note on cross-references

Cross-references are complicated, in that individually 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).[[15]](#footnote-15)

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.[[16]](#footnote-16) 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.

You can limit the books via the command-line parameter *dbgSelectBooks*. The value associated with this must use USX (UBS) book abbreviations, and can identify a single book, a comma-separated list of books, or a range of books in the form *a-b*.

# 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.[[17]](#footnote-17)

**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.[[18]](#footnote-18) I am not sure whether this is still an issue. I did have processing 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**

# 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:



## Creating JAR files

|  |
| --- |
| **Don’t build this as what Intellij IDEA describes as an *artifact* – it won’t work.** |

In order to run the converter from the command line, you need to turn the converter into a JAR file – and in particular one which contains all of the various libraries etc upon which it relies.

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

**And don’t forget to update the version number stored in build.gradle.kts**

The *tasks.jar* section in *build.gradle.kts* is configured to support the build process. Be very careful if changing it – it took a long time to get it to the point where it seems to work, and (perhaps as a result of that) feels rather fragile.[[19]](#footnote-19)

If you *do* make a change to this build file (and as noted above, you will probably want to do so to give revised versions a new version number), a small button appears at the top right of the window after you make the change, and I think you have to click that in order for the change to be taken into account.

You build the JAR not as an artifact, but using the build/jar option from the Gradle task menu. You can find this at the right of the IDEA screen at present – it has an icon which looks like an elephant (but not very much like one). The output turns up in build/libs.

## 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.
* You need to be careful if you are tempted to use things like reflection, or anything else which involves poking around with the package and class structure at runtime. I am actually doing this successfully now, but it took a lot of doing – things which work in the IDE don’t necessarily seem to work in a standalone JAR. In part this seems to be because there is a library which does not get embedded automatically in the JAR even though you mention the need for it in *build.gradle.kts*, and even though other libraries do get included automatically. I did work around this eventually, but can’t now recall full details.

**OSIS**

# OSIS: Detail

The following sections are in no particular order.

Some of the information below implies that we are deviating from OSIS compliance in either the external-facing or the internal-facing OSIS. Some imply that, although compliant, I am having to sacrifice semantic details in the internal-facing OSIS in order to make the rendering acceptable. Both of these I highlight. Others are simply odd ways of working which appear to be required, but which have no further implications.

There are also some more complex issues which are discussed in sections 16 onwards.

## General things to be aware of

### Span-type tags and verse markers

According to the OSIS XSD, things like <hi> are not permitted to contain verse markers. There is some anecdotal evidence to suggest that breaking this particular rule can indeed give rise to problems (see section 16). I suspect there may be similar issues with other span-type tags. Translators may well do things which break this. At present I don’t have any processing in place to address this.

## Arbitrary tweaks with no significant implications for compliance or semantics

### Verse-ends

Sometimes I move verse ends to avoid cross-boundary markup. This should have no visible impact, because I never move verse ends in such a way as to leave canonical text outside of a verse.

### Canonical headings

I replace start-of-chapter canonical heading tags with formatting markup to avoid rendering issues (and end-of-chapter headings too, but for a different reason: these are mentioned below).

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 we will be using the Crosswire mode of *osis2mod* (ie if we are generating a public-facing version of a module), 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 the STEPBible mode of *osis2mod*, 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.

### Speaker tags

I replace ‘speaker’ tags with formatting markup. STEPBible does render speaker tags, but we don’t like the way it does it.

### Acrostic tags

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

Selah tags aren’t rendered well either, and again I replace them 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.

### Avoiding suppression of similar verses

If you have two or more consecutive verses which are similar (identical?), something somewhere suppresses some of them. You might assume this is an unlikely eventuality, but it is actually quite likely to occur, particularly when expanding elisions, because here we end up with a number of consecutive verses all of which contain nothing more than a dash (which we use to persuade users that the verses are not empty by accident).

You can get round this by enclosing the content of alternate verses in an otherwise entirely pointless  
<hi type='normal'>.

### Notes – rendering of callouts

This is slightly confused …

Sometimes the text has a number of consecutive notes / cross-references and separates them by commas – and the commas are randomly rendered or not.

Sometimes the text has a number of consecutive notes / cross-references and does not separate them by commas – but commas are inserted by something before we get as far as the rendered output.

It is not entirely clear what effect we want to achieve. At present, all callouts appear as down-arrows, and where this is the case, there seems to be little point in them being separated by commas. Should we ever get to the point where we use anything else as callouts, the commas might be useful to aid readability.

Anyway, if you have commas and wish to retain them, adding an empty <hi type='normal'> before the note tags seems to do the trick.

And if you do *not* have commas and wish to prevent them from being added spontaneously, adding a single space between the note tags seems to work. (That’s a plain vanilla space – ASCII 0x20. You don’t need a non-breaking space or anything like that.)

### Loss of poetry lines

On at least one text recently, poetry lines were being dropped in an arbitrary manner (ie some poetry lines appeared in the output while others did not).

To address this I am inserting an otherwise entirely pointless <hi type='normal'> before any poetry tag.

### Blank lines

It has latterly become apparent that something somewhere can get screwed up if blank lines appear in the text in certain places.

Blank lines at the very end of a chapter cause the last verse number to come out after the text of the verse so I suppress these.

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.

Experience also suggests that a para:b (<lb>) 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;).

## Changes which may alter the structure or appearance of the text

These will be of particular concern where licence conditions limit what we can do to a text.

### Missing verses

I generate verses to fill any holes before the last verse in a given chapter (but if the chapter stops short of where it *should* stop, I don’t add extra missing verses on the end). This is necessary because *osis2mod* / *JSword* get things wrong otherwise.

### Elisions

I expand elisions out into the individual verses which make them up, and put the entire content of the elision into either the first verse of the set or the last one.

### Tables

Tables are normally extensive enough to run across several verses, which inevitably gives cross-boundary markup. There are several ways to get round this, each of them with its own advantages and disadvantages. Presently, I replace the table by an elision containing all of the verses which make up the table, and place the entire content of the table into one of those verses.

### Plain vanilla paragraphs

Paragraphs are normally enclosing constructs. I change them to self-closing tags which are positioned at the front of the text, so as to reduce the chances of cross-boundary markup. I believe *osis2mod* does this anyway; I get in ahead of it so I can control what happens.

Note that this may be a problem if a paragraph tag is expected to do something to the contained text as a whole (like indenting it – an enclosing tag can change the margins, but a self-closing one cannot).

### Indented paragraphs

Translators commonly use poetry or list tags on the assumption that this will generate indented paragraphs. In fact it does not do so. My guess here – although it is *only* a guess – is that *osis2mod* turns these tags (which enclose their content) into milestone tags (which precede what would otherwise be their content).

I think it does this to avoid cross-boundary markup. However, while an enclosing tag can apply a left and right margin to its content, a milestone tag cannot. Hence at best this can indent the first line.

Given that I believe *osis2mod* does this, I may opt to do it myself anyway for the same reason – to avoid cross-boundary markup.

### Introductory material

Text-, book- and chapter- introductions do not appear in the rendered output.

## Non-compliance

### Enclosing tags [NON-COMPLIANCE]

OSIS requires that poetry and list elements be enclosed in a tag equivalent to HTML’s <ul> tag.

This is problematic, both in that it tends to give rise to cross-verse-boundary markup (which is a problem for reversification and may also be an issue for *osis2mod*), and in that if retained, it gives rise to excessive vertical whitespace when rendered.

We therefore don’t bother with these tags – things seem to cope fine without, and the rendered output looks better. (In fact I have a feeling that even if we retained them, *osis2mod* may itself remove them.)

### Copyright information [NON-COMPLIANCE]

Crosswire support a very limited form of markdown to be used on copyright and related information – there are very few tags, and the ones that there are can be used only in selected fields in the Sword configuration file.

In fact, it appears that many people ignore this, and use HTML instead, and we do this too – it makes it possible to improve the readability of the copyright information, and it also means we can simply pick up the information directly from DBL configuration files when working with DBL data (the metadata for quite a lot of texts contains relevant fields in HTML form).

## Loss of semantics

### Speaker tags [LOSS OF SEMANTICS]

Speaker tags do work, but we don’t like the way they are formatted, so I am replacing them by formatting markup.

### Acrostic tags [LOSS OF SEMANTICS]

Acrostic tags (as both paragraphs and as span-type) tags do work, but we don’t like the way they are formatted, so I am replacing them by formatting markup.

### Selah tags [LOSS OF SEMANTICS]

In theory, selah tags are supposed to be rendered on the current line, right justified. This doesn’t happen, so I am replacing them by an alternative formatting markup. Regrettably this doesn’t achieve the right effect.

### Canonical titles at the ends of psalms [LOSS OF SEMANTICS]

There are a few places where psalms have canonical titles at their ends. 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.

### Notes and cross-references functionality [LOSS OF SEMANTICS]

Footnotes and cross-references do not appear to work outside of verses.

* The *callout* for the note or cross-reference appears, but nothing happens if you hover the mouse over it.
* In some cases you can remedy this by moving the note tag within an adjacent verse (so long as you can decide whether the note belongs with the previous verse or the next one).
* However, there are some cases where this won’t work. For example, I believe I am correct in saying that a note marker within a canonical header is treated as being outside of a verse, and therefore does not work – and you are unlikely to have any obvious place within the first verse to which you can move the note.
* We have no experience of footnotes within introductory material, because at the time of writing we can’t render introductory material.

# Tables

Tables which contain verse markers are a particular problem (except possibly if, throughout the table, the sid and eid for each verse are both in the same cell, but I have yet to encounter a table like that).

Any attempt at all to retain tables which contain verse markers 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.

# Canonical titles at the start of Psalms [POSSIBLE LOSS OF SEMANTICS]

Canonical titles may appear both at the start and at the end of certain psalms. Both give rise to issues. The titles which appear at the *ends* of psalms are discussed in section 15.5.4. The present section discusses titles which appear at the *starts* of psalms. These give rise to much more significant issues.

There are three particular aspects of this which require consideration:

* We may have a text which can be turned into a public module. In this case, we will want to do as little processing of our own as possible – just enough to create something the Crosswire mode of *osis2mod* can cope with. Or we may be creating a STEPBible only module, and need to have something the STEPBible mode of *osis2mod* can cope with. The requirements in the two cases differ.
* There is the question of which versification tradition has been followed in the raw text. In respect of how psalm titles appear in the text, it matters whether you are dealing with something based on the Hebrew tradition, the Latin tradition, etc.
* We need to cater for the foibles of the particular translators. Different people use different tags in different ways.
* And finally there are some things which *osis2mod* just doesn’t seem to like.

At the time of writing, we have been working particularly with GerHFA, so what follows is really conditioned by the issues we are encountering with that (and means that at present we are looking at only one raw-text versification scheme). It would be very nice if the issues and solutions identified here would transfer to other forms of text too (or at least, not flatly contradict them), but that remains to be seen.

## General points

### Replacing psalm:title [LOSS OF SEMANTICS]

It looks as though in some situations (and perhaps all) we shall have to replace title:psalm by formatting markup – most likely hi:italic. Retaining the title:psalm, at least with GerHFA, was giving empty verses, verse numbers in the wrong place, etc.

Note that hi:italic is not rendered in quite the same way as psalm:title. The latter is rendered using a sans serif font, whereas hi:italic simply italicises body text (which uses a serif font).

Note also that psalm:title is automatically set off on a line of its own, so that it does indeed look like a header. hi:italic is not, so a newline needs to be introduced after it (but ideally not if the text already has one there, or there will be excessive vertical whitespace).

### Reversification footnotes

Reversification processing may call for footnotes to be added to canonical titles. It is necessary to suppress these notes, because, as discussed in section 15.5.5, footnotes outside of verses don’t function. Regardless of whether we use title:psalm or hi:italic, the canonical text is treated by *osis2mod* et al as though it *is* outside of a verse.

## Using the Crosswire mode of osis2mod with one of their schemes

I had hoped here that we might simply be able to run with the general structure of the input text. Sadly, this turns out not to be the case.

Different psalms give rise to different kinds of issues, so I shall need to look at each separately.

To give a stake in the ground, GerHFA is well aligned with the Crosswire German versification scheme, which is the one I have chosen to use (it is also well aligned with Luther – I suspect the two are pretty much identical). Or more accurately, the books, the numbers of chapters per book, and the numbers of verses per chapter fit well. I do not consider canonical titles in making this assessment, because I lack details of any expectations Crosswire may have in that respect.

In the following examples, I give a simplified version of the markup which would be derived directly from the GerHFA USX, and then discuss how this needs to be altered (assuming it does) in order to look ok in the rendered output. My aim, as far as possible, has been to align it with the text as it appears at bible.com.

### Ps 3

**<title><verseStart:1/> contents … <verseEnd:1/></title>**

According to the OSIS XSD, title:psalm can contain a verse. In practice, this doesn’t seem to work: indeed retaining the title:psalm at all gives rise to problems.

This therefore needs to be turned into:

**<verseStart:1/><hi:italic> contents … <hi:italic/><verseEnd:1/>\*\***

An additional line break or two may be needed at \*\*. See section 17.1.1 for a discussion of the ordering of the tags here.

### Ps 11

**<title><verseStart:1/>TitleText</title> VerseContents … <verseEnd:1/>**

This needs to turn into

**<verseStart:1/><hi:italic>TitleText</hi:italic/>\*\* VerseContents … <verseEnd:1/>**

An additional line break or two may be needed at \*\*. See section 17.1.1 for a discussion of the ordering of the tags here.

### Ps 51

**<title><verseStart:1/> VerseOneContents … <verseEnd:1/></title>**

**<title><verseStart:2/> VerseTwoContents … <verseEnd:2/></title>**

This needs to turn into

**<verseStart:1/><hi type='italic'> VerseOneContents … </hi><verseEnd:1/>\***

**<verseStart:2/><hi type='italic'> VerseTwoContents … </hi><verseEnd:2/>\*\***

In the GerHFA verse 2 has a footnote. In this particular case (contra section 17.1.2) the footnote does not need to be suppressed, because it will still fall within a verse after processing.

An additional line break may optionally be added at \* (assuming one is not there already) if it is desired to split the dual verse title on to two lines.

And a further line break may be needed at \*\* (again if not already present) to force the title to be set off from the following text.

**Testing**

# Useful places to check output

Tables: Num.1; Josh 12; Ezra 1-2; Neh.7

**\* End of document \***

1. Personally, I don’t think using OSIS as the basis of manual changes is ideal. If we have USX, IMP, etc available to us, applying changes manually to OSIS means we now have two potential different inputs (the OSIS or the original non-OSIS data) and have to worry about whether we have chosen to work with the right one, and whether the two are in step. 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, because otherwise our tweaking would have to be able to cope with multiple different formats. [↑](#footnote-ref-1)
2. This also makes things easier when taking details from things like DBL’s metadata.xml file, which quite often contains the data we need already in HTML form, and not in plain text. [↑](#footnote-ref-2)
3. In this case, the element is assumed to finish when the next milestone is encountered. [↑](#footnote-ref-3)
4. And just to keep you on your toes, chapters may be marked with <chapter> or <div type='chapter'>. [↑](#footnote-ref-4)
5. In other words, where milestone markers are used, the starting-milestone does not normally fall within a tag while the ending milestone (or the implicit position of the ending milestone with start-only texts) falls outside it, or vice-versa. [↑](#footnote-ref-5)
6. I do this because it is an easy way to remove significant numbers of potential cross-boundary issues. Such a change is fine so long as the effect of a para marker appears only at the *start* of the paragraph (ie it perhaps simply introduces preceding whitespace). It is no good if the purpose of the para is to indent an entire block of text, and this is indeed an issue for poetry. However, as I say above, if *I* don’t make these changes, it looks as though *osis2mod* will. [↑](#footnote-ref-6)
7. I haven’t yet come across a case where we also need a country- or script- code, so I haven’t given any thought to how it would be handled. [↑](#footnote-ref-7)
8. Some languages have more than one code (eg deu / ger). In these cases, we have our own preference between the options (which sadly is often not the same as the ISO preference). I think the processing accepts either and then gets it right, but you’d have to check. [↑](#footnote-ref-8)
9. I have to say I’m not keen on these, because they don’t provide any useful information about the text. [↑](#footnote-ref-9)
10. By this, I mean, for instance, that if you are working with a DBL open access text, you should look for another DBL open access text. Ditto a DBL copyright text, etc. In other words, look for texts with similar provenance. [↑](#footnote-ref-10)
11. You can access the content of a JAR file using a standard zip tool. [↑](#footnote-ref-11)
12. Internally, all configuration parameters have names starting *step*. This is a pain. It’s a historical hangover, and I can’t quite convince myself it’s worth the risk of changing it. [↑](#footnote-ref-12)
13. 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 STEPBible configuration language, not of XSLT. [↑](#footnote-ref-13)
14. Restructuring is not something I imagine we will do commonly. It is used to make a text conform to NRSVA structure, which is required if STEPBible’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-14)
15. At one point I felt this was a useful thing to do. Latterly, DIB has suggested it may not be necessary. In any case, it would be relevant only where reversification were physically restructuring the texts, and as explained elsewhere, we no longer anticipate using this option. [↑](#footnote-ref-15)
16. There seems to be an urban myth that STEPBible can handle individual cross-references which are themselves in the form of a collection. It can’t. [↑](#footnote-ref-16)
17. In fact, STEPBible 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 STEPBible 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-17)
18. This was an issue to do with the way the text is rendered: it was not an artefact of the conversion process. [↑](#footnote-ref-18)
19. The build file is also set up to exclude files named \*.RSA, \*.SF and \*.DSA. These have something to do with signing, and stop the JAR from being runnable if they are present. [↑](#footnote-ref-19)