**STEP USX to Sword Converter**

USX to Sword   
text converter



**User and Maintenance Guide**

**Contents**

[1 Introduction 4](#_Toc150968339)

[2 Using the converter – process overview 6](#_Toc150968340)

[3 Installing the tool and configuring the environment 7](#_Toc150968341)

[4 Creating a folder structure 8](#_Toc150968342)

[4.1 The per-text root folder 8](#_Toc150968343)

[4.2 The content of the per-text root folder 8](#_Toc150968344)

[5 Configuration and metadata 9](#_Toc150968345)

[5.1 Overview 9](#_Toc150968346)

[5.2 Detail 9](#_Toc150968347)

[5.3 Using a shared folder structure 10](#_Toc150968348)

[5.4 Taking configuration data from an environment variable 10](#_Toc150968349)

[6 Preprocessing 11](#_Toc150968350)

[6.1 Standalone preprocessing 11](#_Toc150968351)

[6.2 Callable preprocessing 12](#_Toc150968352)

[6.3 Creating a callable preprocessor 13](#_Toc150968353)

[7 Running the converter from the command line 15](#_Toc150968354)

[7.1 Generating a Sword module 15](#_Toc150968355)

[7.2 Evaluating versification schemes 16](#_Toc150968356)

[7.3 Module naming 16](#_Toc150968357)

[7.4 Version numbering and history information 17](#_Toc150968358)

[7.5 Checking the outputs 17](#_Toc150968359)

[7.6 The Sword module 17](#_Toc150968360)

[7.7 Additional information: the TextFeatures folder and the enhanced Sword configuration file 18](#_Toc150968361)

[8 Philosophy and implementation 19](#_Toc150968362)

[8.1 Overview 19](#_Toc150968363)

[8.2 Folder structure 20](#_Toc150968364)

[8.3 Enhanced USX – why and what 21](#_Toc150968365)

[8.4 Reversification 22](#_Toc150968366)

[8.5 Non-compliance 22](#_Toc150968367)

[8.6 Tools etc 23](#_Toc150968368)

[9 A note on debugging 25](#_Toc150968369)

[10 Gotchas and arcane information 26](#_Toc150968370)

[10.1 General 26](#_Toc150968371)

[10.2 Tables 27](#_Toc150968372)

[11 Intellij IDEA 29](#_Toc150968373)

[11.1 Run configurations 29](#_Toc150968374)

[11.2 Artifacts 29](#_Toc150968375)

[11.3 Bugs 30](#_Toc150968376)

[12 Internals 31](#_Toc150968377)

[12.1 org.stepbible.textconverter 31](#_Toc150968378)

[12.2 org.stepbible.textconverter.support 32](#_Toc150968379)

[13 Shared configuration files 34](#_Toc150968380)

[14 Conversion policy 35](#_Toc150968381)

# Introduction

This document talks about the STEP Text Converter. This is a tool which takes a Bible text in ‘any’ format and generates from it a Sword module which can be used in STEP.

(At the time of writing, ‘any’ here actually means just USX.[[1]](#footnote-1) The architecture is aimed at supporting other things too, but additional development work would be required. Having said which, note that as things stand, we can also indirectly support USFM, because UBS’s Paratext tool can be used to turn USFM into USX before we start work.)

*Some additional work may be needed to handle extended tagging. I have a framework in place to address this where the extended tagging is being applied as part of the process starting from USX. However, we are also considering applying tagging to third party data for which we have only OSIS or only the module itself.*

The first part of this document is concerned with *using* the tool. Later sections go into some detail on the internals.

The overall flow of control is summarised below, although if your only interest is to use the tool, you don’t really need to concern yourself too much with this.

|  |
| --- |
| USX  OSIS  Sword  EnhancedUSX  Preprocessed USX  Reversification |

And here’s a brief explanation – sufficient, I hope, that if your only interest is to be able to use the program, you can spare yourself the hassle of reading about the internals.

* Each of the boxes above reflects a file structure based upon a particular syntax. The converter progressively changes one file format to another.
* USX and OSIS are ‘official’ XML dialects. USX is a living standard (ie it continues to be subject to change), and is controlled by UBS. OSIS is moribund (ie it is no longer subject to change).
* The Sword format is proprietary to Crosswire (and we have no official documentation for it, although a certain amount can be gleaned from reading the source code of the tools which work with it).
* Enhanced USX is our own internal XML format. It is there to overcome certain shortcomings of USX and to make later processing easier. If you are simply *using* the tool (as opposed to understanding and maintaining it), you do not need to be acquainted with enhanced USX. (It is discussed further in the sections dealing with internals.)
* The start and end of the above chain are dictated for us. It is a pragmatic decision to take USX as our starting point, because these days almost all the texts we see are in USX format. And the decision to create Sword modules was taking at the very inception of the STEP project, and is therefore pretty much set in stone.
* The use of Sword modules in turn forces upon us the need to convert everything to OSIS: the Sword format is proprietary to Crosswire, so we are reliant upon a Crosswire-supplied tool (osis2mod[[2]](#footnote-2)) to create modules, and this requires OSIS as input.

# CAVEATS

The processing – and this document – was originally set up on the assumption that we would take a format other than OSIS, and then convert it to OSIS before then using osis2mod to turn it into a Sword module.

# Using the converter – process overview

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

# Installing the tool and configuring the environment

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

* You need to have the file TextConverter.jar stored somewhere; and depending upon how you choose to run it, you may need to have it in your classpath.
* You need to have Java 18 or later installed.
* You need a copy of the Crosswire *osis2mod* program (see <http://dl.thehellings.com/sword-utils/> for the Windows versions).
* You also need a copy of our own *osis2mod* variant (which I think is currently available only in a Windows version). I can’t give the actual name for that, because I have a feeling it may change.
* I think (but am not 100% sure) that both of these rely upon DLLs – and unfortunately, the DLLs they use in the two cases have the same names. I therefore strongly recommend that you put the Crosswire version, along with its DLLs in one folder, and our proprietary version along with its DLLs in another.[[3]](#footnote-3) You will also need to set up an environment variable indicating where they can be found. The variable should be called *StepTextConverterParameters*, and you need it to contain the following (which you should assume to be case-sensitive):

stepStepOsis2ModFolderPath=somePath; stepCrosswireOsis2ModFolderPath=someOtherPath

This is actually the syntax used for specifying configuration parameters more generally, and indeed this environment variable can be used for more than giving just these two settings. Configuration information in general, and this environment in particular, are discussed further in section 5. Note, incidentally, the repeated ‘step’ at the start of the first setting. Internally, all configuration parameters have names starting with ‘step’; and here the parameter gives the location of the STEP version of the osis2mod program. Hence ‘stepStep’.

# Creating a folder structure

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

## The per-text root folder

* You need to create a separate root folder for each text.
* It doesn’t matter where you store this folder, although it may be convenient to co-locate folders for related texts (eg all texts from the same publisher) under a common ancestor so they readily can share configuration information. We look at this idea of shared data further in section 5.3.
* Give the folder a name like eg *Text\_eng\_XYZ*, where in this example *eng* is the two- or three- character language code for the text, and *XYZ* is the abbreviation for the name of the Bible, the two parts being separated by an underscore. My convention is to give the former in lower case, and the latter in whatever format seems to make sense. (The module name will be based upon this, but upper-cases the language code and drops it altogether in some cases.)
* Very often the translators will have provided an abbreviated name for you.[[4]](#footnote-4) Otherwise you will have to make one up. It needs to be unique (so if one has been supplied, but that clashes with something else, you will have to make one up). There is no limit on the length of the abbreviation, but you should try to keep it short, because it is used in module names, and these appear in places where screen real-estate is at a premium.

## The content of the per-text root folder

* Within the per-text root folder, you need to create a folder called *Raw…* and a folder called *Metadata*.
* If you are taking USX as input, the *Raw* folder needs to be called *RawUsx*, and you need to store the USX input files in *RawUsx*, one scripture book per file. It doesn’t matter what the files are called (except that they must be \*.usx), because the processing looks at the file *content*, not at the file *name* to determine which book each contains.
* If you are using existing OSIS as input, the *Raw* folder needs to be called *RawOsis*, and you need to store the OSIS file within it. It doesn’t matter what it’s called, except that it must be \*.xml.
* The *Metadata* folder is used to hold configuration information. We look at this in section 5.
* The processing itself stores extra folders and files within the per-text root folder.

# Configuration and metadata

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

## Overview

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

Among other things, the metadata has to cater for :-

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

This implies a very large amount of configuration data. Most of this data resides in the *Resources* section of the converter JAR file, but fortunately normally only a very small proportion of it needs tailoring to a particular text. In general, therefore, the amount of work involved is quite modest.

## Detail

* Copy the file *step.conf* from the *Resources* section of the converter JAR file (you can access a JAR file using an ordinary zip tool) and store it in your *Metadata* folder.
* Follow the instructions within the file to tailor it for your text, or follow the example of a file which already exists.
* If the text you are processing came from DBL and you have DBL’s *metadata.xml* file available (and wish to pick up data straight from there rather than transcribing it all yourself), store that file in the *Metadata* folder.[[5]](#footnote-5)
* If you are working with a DBL text, it can also be useful to store the *license.xml* file in the *Metadata* folder. This offers the possibility of using a tool to extract administrative information to monitor licence expiry.

## Using a shared folder structure

The configuration data supports an ‘include’ facility which lets you include one configuration file from within another (any number of files, nested to any reasonable depth). Such files can be specified as lying within the JAR file and located relative to its *Resources* section (you give a name starting $common/); relative to the *Metadata* folder of the text you are processing (you give a name starting $metadata/); or relative to the text’s root folder (you give a name staring $root/). Or you can give an absolute path. Or, if you give a path which fits none of these patterns, the path is assumed to be relative to the file which contains the $include directive. In each case, you can then use all of the standard directory markers – . and .. and /

This gives an easy way of sharing data among texts. For example, if you have a file containing Hausa translations for frequently used footnote texts, and if all Hausa Bible texts are stored in folders below a common parent, you can put the translations into the parent folder and then easily access the one copy from all of the individual Bible folders.

## Taking configuration data from an environment variable

Reference was made in section 3 *StepTextConverterParameters* environment variable. This can be used for more than just the purpose discussed there – you can store in it any settings you like. With one exception (it doesn’t cater for defining external metadata sources), you can include in it any settings you like – just give a list of configuration directives separated by semicolons.[[6]](#footnote-6)

One setting you may find particularly useful is

stepTextConverterDataRoot=someFolderPath

If you store all of your texts under some common folder, this gives that folder. You can then give locations relative to that for each of your texts. Thus, for example, you have stored all texts under *C:\SomeComplexStructure\Texts*, and you have text folders *Text\_eng\_KJV* and *Text\_eng\_NIV*, when you run the converter you can give their root folders as *Text\_eng\_KJV* and *Text\_eng\_NIV*, rather than as the full paths including *C:\SomeComplexStructure\Texts.* That way, if you need to change stuff in future, you can do so more easily.

# Preprocessing

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

***It is run automatically every time the converter runs.***

USX being a rather complex standard (and at the same time lacking facilities which translators commonly seem to require), it quite often happens that we receive texts which are incorrect in some manner (or at the very least, which do not fit with our processing).

The manner in which such texts deviate differs from text to text. This means it is not really feasible to extend the converter to handle all of them – to do so would produce something even more baroque than we already have.

You *can* simply apply modifications manually. However, this may be time-consuming if there are many of them, and such modifications will be lost if the issues remain in any new version of the text we receive in future.

I have therefore made provision for you to write your own text-specific preprocessing, and to have this automatically picked up and applied by the converter before it starts its work proper.

I support two different forms of preprocessing. In one, the preprocessor is run standalone. In this version, it reads the raw USX files and creates new files as necessary. In the other, the preprocessor effectively becomes part of the converter for the duration of the run, and works in-memory – it is passed a document object model instance, and returns a revised version of the module. These are described in detail below.

## Standalone preprocessing

In standalone preprocessing, you create a runnable JAR, or any other program (.jar, .js, .exe, .py, .bat etc) which is capable of taking arguments from the command line.

You inform the converter of the requirement to run the preprocessor by setting the configuration parameter *stepRunnablePreprocessorFilePath* to give the path to the program.

Particularly with scripting languages, you may also need to give details of how to run the interpreter, to which end you can use the configuration parameter *stepRunnablePreprocessorCommandPrefix*.

Thus, for example, if you have a Perl script *C:\Users\Me\Documents\perl.pl*, you give *stepRunnablePreprocessorFilePath* as *C:\Users\Me\Documents\perl.pl* and *stepRunnablePreprocessorCommandPrefix* as *perl*, and the converter will run this as  
  
 perl "C:\Users\Me\Documents\perl.pl" <args>

There is no need to use *stepRunnablePreprocessorCommandPrefix* with .jar files, .py files or .js files, unless you need to do something odd: the converter will assume *java -jar*, *python* or *node.js* respectively.

*stepRunnablePreprocessorFilePath* works in much the same way as the $include configuration statements discussed in section 5, so you can give the path as relative to the root folder of the text (using $root), relative to the metadata folder, etc.

Including arguments, a command line might look like:

C:\Users\Me\Documents\preprocessor.exe <outputFolderPath> <inputFolderPath> [<bookList>]

The arguments are as follows:

* *inputFolderPath*: The folder in which the raw USX files are stored. The processing should run over all \*.usx files in this folder (or selected ones – see the discussion of *bookList* below), and apply modifications as necessary. Anything other than \*.usx should be ignored.
* *outputFolderPath*: The folder in which to store output files (all of which must be valid USX). There is no need to store a file in this folder unless the processing has changed it (but no *harm* in storing it regardless). Files should be stored under the same name as the corresponding input file. The output folder will be empty at the time the preprocessing is invoked, so there is no need to clear it before starting work.
* *bookList*: This may or may not appear, and even if it does, there is no requirement for the preprocessor to take any notice of it. It gives a list of UBS (USX) book abbreviations which are the ones which the preprocessor must handle on this run. Any others can be ignored for the sake of speeding things up and reducing file io, but there is no problem if you choose to process them regardless.

It is unlikely that you will want to create in the output folder a file for which there is no input counterpart, but you can do so if there is a need for it. In this case you can give it any name you like, but it must have the .usx extension.

The preprocessor can also prevent a given file from being processed by the converter, by creating in the output folder an empty file of the given name. Again, though, I should imagine this is an unlikely eventuality.

When carrying out the actual conversion, the converter will pick up files from the output folder in preference to corresponding files in the input folder, where both exist.

There are several advantages to using a standalone preprocessor. In general it is easy to debug, because you can indeed run it standalone to test it. You also need do less work in support of validation. The converter validates its own output against the input which it used. With a standalone preprocessor, you do not need to do anything to make this work. This is not necessarily true for the callable preprocessor discussed in the next section.

There are downsides, however, of which probably the main one is the increased file io in creating revised copies of the input files.

## Callable preprocessing

With callable preprocessing, you supply an external JAR file which conforms to certain rules, and this is incorporated into the converter temporarily at runtime.

To tell the converter it needs to use it, you use the configuration parameter *stepCallablePreprocessorFilePath* to give the path to the JAR. See the discussion of standalone processing for details of the various syntactic sweeteners you can use for this.

The easiest way to describe what the code in this JAR should look like is to give an example:

package **org.stepbible.preprocessor** // You must use this package.

. . .

**fun main(args: Array<String>) {** } // See discussion below.

class Preprocessor

{

**fun getTextForValidation (text: String): String**

{

return text.replace("¶", "")

}

**fun preprocess (doc: Document): List<String>?**

{

val textNodes = findAllTextNodes(doc)

textNodes.filter { "¶" in it.textContent } .forEach {

it.textContent = it.textContent.replace("¶", "")

}

return null

}

// Anything else you need.

}

Strictly speaking *fun main ()* above is not needed, because nothing will attempt to call the main entry point. However I had problems in Intellij IDEA without it – it seemed to be difficult to get the JAR file set up correctly, and syntax highlighting in the editor wasn’t working correctly.

*preprocess* is passed a document and updates it. In the example here, I’m stripping paragraph markers from all text nodes. *findAllTextNodes* is part of the *Anything else you need* referred toabove. *preprocess* can return error, warning and information messages via its output. Each should be prefixed by ‘Error: ’, ‘Warning: ’ or ‘INFORMATION: ’. They can appear in any order, and the different flavours can be interleaved with each other if that is easier.

*getTextForValidation* is a support method for the converter’s validation processing. The validation processing compares the canonical content of verses in the raw USX with that in the enhanced USX. Except that it takes the canonical content of the *raw* version and passes it to *getTextForValidation* and actually works with the value returned by that. If, as here, the preprocessor has altered the canonical content of verses, *getTextForValidation* should return the value which the preprocessor will have assigned. On runs where the preprocessor has *not* changed the canonical content, it should simply return its argument.

The advantage of this form of preprocessing is that it fits quite neatly into the overall conversion process, requires no additional io, and does not have the complications inherent in standalone processing of having to locate all potentially relevant files.

The downsides are the difficulty of debugging the code, and the need for *getTextForValidation*. In the above example it was easy to implement, but it is not hard to imagine scenarios in which it would be appreciably more complicated. In these, it might well be better to resort to the standalone preprocessing, because there (trust me …) any work done by the preprocessor is no an issue, and there is therefore no need for a *getTextForValidation*.

## Creating a callable preprocessor

The mechanics of creating a callable preprocessor using Intellij IDEA have proved somewhat challenging, so I give details of what I think is needed below. Take this with a pinch of salt; even now I’m not sure that this is complete and correct, nor that there aren’t things below which are unnecessary.

* Make a Kotlin project for the preprocessor. Have IDEA create the sample main for you – we don’t need the main function for the processing, but without it, IDEA seems to go wrong: syntax highlighting doesn’t work, and when you build the JAR as an artifact, the relevant code doesn’t end up in it.
* Create the code according to the template above.
* Go to *File/ProjectStructure/Artifacts*, hit the ‘+’ at the top left and then select *JAR* and then *FromModulesWithDependencies*.
* In the resulting window ... Against *Modules*, select the item with ‘.main’ against it. Set *MainClass* (even though we’re never going to use the main method). Make sure *ExtractToTheTargetJar* is selected.
* Build the artifact. It will turn up in the ‘out’ folder, from where you can move it to anywhere convenient.
* Don’t forget to rebuild it and copy it again each time you need to change it.

# 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.***

## Generating a Sword module

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

**rootFolder** is the path to the root folder for the given text (for example the *eng\_XYZ* folder mentioned earlier). If it’s an absolute path, that’s what’s used. If it’s relative, the system attempts to locate it relative to the current working directory. If that doesn’t work, it looks for the setting stepTextConverterDataRoot discussed in section 5.4. This parameter is *required*.

**runType** may be *Release / MajorRelease / MinorRelease* (generate a release version of the module); *EvalOnly* (generate a version for evaluation only); or any of the alternative names listed in the help for the program. These latter alternatives represent test flavours which the system has been set up to handle. Different kinds of tests may apply different kinds of instrumentation etc. Version numbers and history information are updated only on release runs. You can force the question of whether this is a major or minor release using *MajorRelease* or *MinorRelease*. If you just use *Release*, the processing will decide for itself whether this is a major or a minor release. See the discussion of version numbering in section 7.4 for an explanation of how the choice affects version numbers and history information.

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

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

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

**summariseTextFeaturesOnly** If present, processing does not run to completion. Rather it stops as soon as it has generated output summarising the kinds of features present in the text. This parameter is optional and defaults to *No*.

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

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

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

## Evaluating versification schemes

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

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

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

If you are not intending to apply reversification / not using our proprietary version of osis2mod, then you will need to use this output to select an appropriate scheme for use when carrying out the conversion process, and must then set stepVersificationScheme accordingly before generating a module.

Note that texts relatively seldom fit any ‘standard’ scheme exactly, and the Crosswire version of osis2mod may itself restructure a text which does not precisely fit the selected scheme.

## Module naming

The basic name of a module looks like *DeuHFA* where *deu* is the 3-character language code, and *HFA* the abbreviated name of the text.

Modules always come in one of two flavours – ones which can be used only within STEPBible, and ones which might reasonably be made publicly accessible. The module name has a suffix added to it to indicate which of these two categories the module falls into – *DeuHFA\_sb* for general consumption, or *DeuHFA\_sbOnly* where it is limited to being used within STEP. (The \_*sb* in the former case is used purely for disambiguation purposes – the module may have been built by enhancing third-party data, and we need to make sure we do not generate a module whose name clashes with any name used by the third party.)

At the time of writing, \_*sbOnly* modules are generated if the module is encrypted or if it has been processed via our own version of *osis2mod*. In both cases, the module will not work outside of STEP.

On anything other than a release run, a timestamp is added to the module name, giving something like ***DeuHFA\_****EvalOnly\_0927T1908*. This makes it possible to keep several versions of the module lying around while carrying out testing and evaluation.

## Version numbering and history information

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

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

I have therefore taken an easy way out: we treat as major any update occasioned by the release of a revised version of the source package by the text suppliers, no matter what the revision may have been; and we treat as minor any release occasioned by a revision to the conversion software. There is absolutely no reason why this should be appropriate – a change to the source package may have zero impact, and a change to the software may have a huge impact – but at least it’s easy to be consistent.

The history information has to end up in the Sword configuration file. This is replaced on each run (and will not exist before the first run). It is held, therefore, at the end of the step.conf file, in an area updated by the processing on each run. You can edit this information in the normal way should you need to. I recommend treating this as plain text, though, and not relying upon any of the normal configuration interpolation facilities, because the history lines are deleted from the file and replaced on each run, and so if you have a line which relies upon interpolation, it will be lost on the next run.

History and version information is updated only on release runs.

It is up to you to maintain *Obsoletes* information manually. There is a *copyAsIs* directive described in the head-of-file comments to *step.conf* which you can use to add appropriate directives to your config file.

## Checking the outputs

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

## The Sword module

The Sword module is created as a zip file in the *Sword* subfolder. Its name follows the module name as discussed in section 7.3. On release runs, a further zip file is stored in the root folder for the text, containing this zip file and supporting data (for example the OSIS used to create it). This zip file is intended to support any investigations or reconstruction of the text which may subsequently turn out to be necessary.

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

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

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

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

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

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

# Philosophy and implementation

## Overview

The converter is a collection of Kotlin code which takes Bible texts in a variety of formats, creates from them an ‘equivalent’ OSIS representation, and then uses an external program (*osis2mod* [[7]](#footnote-7)) to turn this into Sword modules.

In fact, reference to a ‘variety’ of input formats is presently misleading. The philosophy of the converter was based upon the assumption that we might want to be able to process a variety of different input types, and indeed might wish to generate a variety of different output types. However, at present, the converter accepts only a single form of input (USX), and creates only a single form of output (OSIS, and thence, via *osis2mod*, Sword modules).[[8]](#footnote-8)

This generic philosophy remains apparent really only in the fact that I convert the input to an intermediate form (‘enhanced USX’), and then generate the output from that. For convenience, I repeat below the schematic of this process from section 1:

|  |
| --- |
| USX  OSIS  Sword  EnhancedUSX  Preprocessed USX  Reversification |

The boxes in the diagram represent different file formats, and the overall process involves progressively changing from one format to another, with the solid arrows indicating Java processing within the converter, and the dotted arrow the action of *osis2mod*.

Briefly working through these steps:

* USX is what it is. It is a perfectly workable standard, but suffers from certain shortcomings and complications.
* Pre-processed USX is still in USX format. I support a pre-processing step because USX is complicated enough that translators quite often get it wrong. There are common ways in which such errors turn up, and processing built into the converter handles these common issues. But where a text gets things wrong in a manner unique to itself (fortunately only a few do), I allow for a separate external pre-processing step to make good the damage.[[9]](#footnote-9)
* Enhanced USX I look at in detail in section 8.3.
* OSIS is forced upon us because we have to use the Crosswire *osis2mod* tool to create Sword modules, and this requires OSIS as input. Unfortunately the mapping between USX and OSIS is far from 1:1.[[10]](#footnote-10)

## Folder structure

This section shows the folders which appear within the root folder for each text. Folders highlighted in orange below have to be created and populated manually before processing. The others are created and managed during the conversion process.

|  |  |  |
| --- | --- | --- |
|  | EnhancedUsx | The processing creates this to hold the enhanced USX generated in the course of processing. |
|  | **Metadata** | You need to create this folder, and place into it, at the very least, *step.conf*. See section 5 for more details. |
|  | Osis | The processing creates this to hold the generated OSIS. |
|  | **RawUsx** | You need to create this folder, and store the raw USX within it, one file per book, each with the extension .usx. The names of the files don’t matter, because the processing looks at the file content to determine what each contains. |
|  | Sword | The processing creates this and stores under it the file structure needed in order to produce a module. In particular, immediately within this folder it generates a zip file which is the module itself. |
|  | TextFeatures | The processing creates this and stores within it certain potentially useful information about the text. |
|  | converterLog.txt | At the end of processing, contains a summary of any issues – including the main ones from *osis2mod*. |
|  | osis2ModLog.txt | At the end of processing, contains a summary of any issues identified by *osis2mod*. |

## Enhanced USX – why and what

I decided early on that it would be useful to have some intermediate format – mainly because at that time I thought we might have to cope with more than one flavour of input, and more than one flavour of output. The use of an intermediate would have brought with it the usual advantage that if we wished to deal with *m* different kinds of input and *n* different kinds of output, we should need only *m* front ends and *n* back ends, rather than having to create *m* × *n* different converters. As things stand, with only a single flavour of each, this advantage does not apply. However, I have deemed it useful to have an intermediate representation anyway, to fix certain shortcomings of USX and to carry out certain processing which would probably be useful regardless of what form of downstream processing is required.

In the end, I could see little point in creating yet another entirely new markup scheme, and since USX was certainly a reasonably starting off point, I decided that I would use my own extended version of USX. This adds certain new tags (and / or creates revised tags to replace existing ones) and adds various attributes to record error or debug information or to simplify later processing).[[11]](#footnote-11)

It is difficult to express concisely the sorts of things which justify this being referred to as ‘enhanced’, but a few examples may help :-

**USX versions**

USX comes in a variety of flavours (version levels), and the different versions differ in some very significant ways. It is convenient to collapse them all to a single form (ie to enhanced USX) in order to hide this complexity from subsequent processing.

**Lack of verse-end tags**

Some versions of USX lack verse-end tags. Enhanced USX inserts them.

**Milestone markup**

USX makes heavy use of milestone tags, particularly to mark the boundaries of books, chapters and verses. This is intended to make it possible for formatting and semantic markup to run across these boundaries, and translators often make use of this.

In fact, for books and chapters, cross-boundary markup is – in my experience – never an issue; and since it is much easier to work with enclosing tags rather than milestone tags, I convert book- and chapter- tags to enclosing tags.

For verses, things are much more complicated, because here markup often *does* run across boundaries. This must surely be a problem for very many tools which have to identify and work with the content of verses. Certainly for us it can be a problem with reversification (we sometimes have to move verses to new locations, and that’s difficult if markup starts or ends within the verse); and the *osis2mod* tool also complains occasionally if it does not like the way things are structured (although whether it gets things wrong in such cases, I’m not sure).

In general, experience suggests that we seldom have a *major* problem here, so I don’t attempt the (usually impossible) task of converting all verse markup to use enclosing tags. But I do try to adjust the *position* of verse milestone tags (naturally without changing their canonical content) so that cross-boundary markup is less likely to occur.[[12]](#footnote-12)

## Reversification

Not all Bibles are created equal. All of the ones with which we are likely to work are split up into chapters and verses, but not all of them are divided up in the same way as KJV(A) (or NRSV(A), which is KJV’s modern equivalent, at least as far as structure goes). This is a problem because very many study tools assume KJV versification; and it is also a problem if you wish to compare different Bibles, because a given underlying chunk of Greek or Hebrew text may give rise to vernacular text carrying one verse number in one Bible, and a different one in another.

To get round this, the converter offers the option of restructuring the enhanced USX to bring it in line with NRSV(A) before using it to create the OSIS. Usually, the greater part of a given text is left unaltered by this activity, or at most has extra footnotes added to explain the manner in which a given verse may be handled in other Bibles, but sometimes subverses may be merged, verses may be renumbered (sometimes leaving them in the same physical location, as when the last verse in one chapter ends up as the first of the next, but sometimes physically moving them); and in a few cases, entire chapters may be moved elsewhere – even to deutero-canonical books which were not present in the raw text.

My personal view has been for a long time that this approach is misplaced, because the result is a ‘familiar’ text with unfamiliar verse numbering, which I felt would not readily be accepted by our target audience. Rather, I felt that we needed to retain the text in its original form, and then change the added value processing to make it work with this structure – except that this sounded like a very major undertaking.

Latterly, the need to retain the original appearance of the text has become rather more pressing, with the realisation that when we are dealing with copyright texts, licensing conditions may preclude us from applying the kinds of changes reversification requires (or alternatively may force us to look for ways of restructuring without appearing to have done so).

At present, the preferred approach to reversification is rather up in the air – we are investigating alternatives involving generating our own version of osis2mod and possibly (I believe) other changes to the STEP platform.

However, as of now, the jury is out on all of these issues.

## Non-compliance

**OSIS XSD compliance**

In order to be able to supply modules or OSIS to Crosswire, the OSIS involved has to validate successfully against the OSIS XSD. There are various things we are doing which mean that sadly we cannot pass this particular test, but there appears to be no way of being compliant while, at the same time, generating modules which work for us. One particular example of non-compliance is given in the next subsection. However, DIB has pointed out that a fair proportion of the texts with which we work are copyrighted, and compliance with these is not really relevant, because we can’t make them available to other people anyway. I guess the counter-argument to this is that by developing the conversion process with this in mind, we are building non-compliance into it, and therefore putting ourselves into the position where we cannot make *any* of our material available, copyright or not.

**Lists and poetry**

USX does not enclose bullet-point lists and poetry within the equivalent of HTML’s <ul> tags. Officially OSIS requires them. However, it is fiddly adding them (particularly if we want to avoid cross-boundary markup), and things seem to work well without. (In fact, if the bracketing constructs are added, we end up with excessive vertical whitespace when text is rendered.) In view of this, I do not generate these constructs. Unfortunately this means that our output will necessarily not validate against the OSIS XSD,

**Metadata**

A number of the items which end up in the Sword config file subsequently appear on the STEP copyright page for the text. Where these items are picked up automatically (eg from DBL’s *metadata.xml*), these may be in HTML format as supplied to us; and even where not picked up automatically (and therefore generated manually), it is often quite convenient to use HTML format in order to obtain a satisfactory appearance.

Strictly we should not be doing this – Sword uses its own markdown for this kind of material, and really we should be limiting ourselves to this. However, the temptation to use HTML is somewhat irresistible since it works (for us, at least, with STEP being browser-based); it is more flexible; and no intervention is needed when picking data up from third parties.

## Tools etc

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

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

# A note on debugging

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

At the top of the main function in Main.kt is a statement Dbg.setBooksToBeProcessed("…"), which can be used to limit processing to an individual book, to a list of books, to a range of books, etc.

# Gotchas and arcane information

## General

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 very 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 this information.[[13]](#footnote-13)

**Lists and poetry**: As mentioned elsewhere, OSIS compliance requires that bullet point elements and poetry lines be enclosed in bracketing tags equivalent to HTML’s <ul> tag – and I am not generating the bracketing because it is difficult to do so, complicates the issue of cross-boundary markup, and also gives excessive vertical whitespace at the start and end of lists or sections of poetry when rendered. I suppose it is possible that the issue of excessive vertical whitespace might be addressed by making changes to STEP’s internal style sheet as mentioned in footnote 13.

**Special formatting, and also lists and poetry**: OSIS has no simple constructs to support things like indented or right justified paragraphs. The SIL recommendations suggest that these be handled using <list> and <item> in various combinations (ie as bullet point lists). However, this is not really a viable solution for us – bullet point lists do, as the name implies, end up with bullet points, which is seldom (never?) what the translators had in mind. The only suitable approach I can find is to use the <l> poetry line tag (but without the bracketing <lg> tag which OSIS compliance demands – see previous paragraph). Even then, this does not really give the rendering which the translators will have had in mind.

**Selah**: In theory, the style='qs' (Selah) tag is supposed to render the enclosed text (which, at least in English Bibles is normally the word ‘Selah’) 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.

**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.

**Blank lines**: It has latterly become apparent that something somewhere can get screwed up if blank lines appear in the text – 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;).

**Elisions**: The processing requires that all individual verses which appear in the target versification scheme (typically NRSV(A)) actually exist in their own right. Elided verses are therefore automatically split out by the processing: if an elision covers vv1-10, we end up with blank verses numbers 1, 2, … 9, and then a single large verse, numbered 10, containing the full text of the elision.

**Tables**: Tables are a big enough problem to warrant a section of their own – see section 10.2.

**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 an empty char-type markup on alternative verses of this kind, so that no two adjacent verses have the same content.

**RTL**: At present, RTL texts have to be rendered in verse-per-line format, or else the text comes out in the wrong order.[[14]](#footnote-14) The processing *should* take account of this automatically, but you can 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.

## Tables

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

I currently support two approaches to handling this (selectable via a compile-time switch): presently the second option is used.

Option 1: I retain the verse markup, but remove the table markup, instead introducing non-breaking spaces between the columns. This has the advantage that all verses retain their correct content, but the disadvantage that the tabular appearance will normally largely be lost.

Option 2: 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 vv1-9 empty, and the entire table in v10. This retains the tabular appearance, but at the cost of the verses no longer having their original content (with knock-on implications for added value such as verse vocabulary, interlinear, etc).

**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.

# Intellij IDEA

## Run configurations

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



## Artifacts

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

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

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



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

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



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

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

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

## Bugs

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

# Internals

The converter has gone through many incarnations. The latest is probably the most compact to date, but the conversion process is more complex than might be expected, and there is therefore a lot of it code, a fair bit of it pretty complicated.

I have recently transitioned to Kotlin from Java, on the grounds that Kotlin is much less verbose than Java and more uniformly structured when it comes to things like functional programming, which forms the bulk of the conversion process.

The code is split into two separate portions – ‘top level’ code, in the org.stepbible.textconverter package, and library code in a number of different packages under org.stepbible.textconverter.support. Ideally, I guess, the latter should really be turned into a pukka stand-alone library, but I haven’t worked out how to use the IDE to achieve that.

All files contain extensive documentation, so I restrict myself here to summarising what to look for where.

## org.stepbible.textconverter

* **CompileTimeConfig**: Flag settings which control the processing but which I have not thought worth putting into separate configuration files.
* **CrossReferenceProcessor:** Parses, validates and possibly (eg in light of reversification) remaps cross-references.
* **DataSummary:** Holds information about the text – which tags it uses etc. The processing does not make use of this data, but I’m gathering it on the off chance it ever becomes useful for things like trying to find texts with particular characteristics.
* **DbgController:** Lets debugging function like any other top-level aspect of the conversion processing, giving a convenient location to specify command-line parameters etc.
* **EmptyVerseHandler:** Empty verses turn up under a variety of different circumstances, and I felt it was convenient to have a common place for dealing with them.
* **GeneralEnvironmentHandler:** Rather a mish-mash. Handles eg config information which needs to be analysed and processed very early in proceedings.
* **Main:** The main program.
* **Osis2ModInterface:** Handles those aspects of the processing which are specific to the particular versions of osis2mod we are using (Crosswire’s or our own).
* **PreprocessorHandler:** Handles the interface to the various forms of pre-processing.
* **RepositoryPackageHandler:** Creates the package which provides data to be lodged in the server repository.
* **ReversificationData:** Reads and stores reversification data. Note that this is used even on non-reversification runs, because the reversification data includes some information needed regardless of whether reversification is actually applied.
* **ReversificationRuleEvaluator:** Evaluates the rules which determine which reversification data applies to a particular text.
* **RuntimeConfig:** Holds flags which control the runtime activity, but (unlike most of the ‘genuine’ configuration information) cannot be determined until runtime.
* **TestController:** Handles things like changing module names when we are performing tests (so that we can have several versions of a module lying around at the same time without them overwriting one another), saving information which may be useful for investigations, etc.
* **TextConverterController**: The overall controller for the conversion process.
* **TextControllerEnhancedUsxValidator:** Applies various types of validation to the enhanced USX. In particular, this attempts to give some measure of confidence that things have not been screwed up in the course of processing – ie that verses still contain the right content.
* **TextControllerFeatureSummaryGenerator:** Creates the files which record ‘interesting’ aspects of the text being processed in case we need to know which texts do what.
* **TextConverterProcessorBase**: A base class from which individual processing steps inherit.
* **TextConverterProcessorEnhancedUsxToOsis**: Converts enhanced USX to OSIS.
* **TextConverterProcessorEvaluateVersificationSchemes**: Useful mainly when not reversifying. Evaluates the versification structure inherent in the raw USX against the various schemes supported by osis2mod, and recommends the one to use.
* **TextConverterProcessorOsisToSword**: Creates a Sword module from OSIS,
* **TextConverterProcessorReversification**: Controls reversification.
* **TextConverterProcessorUsxToEnhancedUsx1**: Handles aspects of the creation of enhanced USX which have to have been addressed before reversification, assuming reversification is applied.
* **TextConverterProcessorUsxToEnhancedUsx2**: Handles aspects of the creation of enhanced USX which have to have been addressed after reversification, assuming reversification is applied. The overall sequence is therefore TextConverterProcessorUsxToEnhancedUsx1 (always), followed by TextConverterProcessorReversification (sometimes), followed by TextConverterProcessorUsxToEnhancedUsx2 (sometimes).
* **TextConverterProcessorVLToRawUsx**: Converts Verse-per-Line (VL) format to USX. This is not integrated properly into the processing chain because it appears there is no common understanding of what constitutes VL, and therefore this is likely to need changing on an ad hoc basis.
* **TextConverterTaggingHandler**: At the time of writing, only in skeleton form. Intended to handle extended tagging, once the details have been firmed up.
* **TextConverterVersificationHealthCheck**: Checks that the chapter/verse structure is valid (eg that verses always appear within chapters).
* **VersionAndHistoryHandler**: Works out version numbers and maintains history information.

## org.stepbible.textconverter.support

Each of the elements below is a package, each containing various source files. I summarise only the packages here, not their contents.

* **bibledetails**: Holds details of the various Bibles – which books and chapters they contain, for instance.
* **commandlineprocessor**: What it says on the tin.
* **configdata**: Anything obtained from config files; details of standard file names and locations; etc.
* **debug**: Debug and logging.
* **iso:** Language codes, locales, etc.
* **miscellaneous**: Dom-related code; locale-handling; string- and file- handling; etc, etc.
* **ref:** Scripture reference handling.
* **shared**: Any data which needs to be shared and which has no other natural home.
* **references**: Everything to do with parsing, storing and outputting scripture references.
* **stepexception**: STEP-specific exceptions.
* **usx**: Information about USX tags – which contain canonical text, etc.

# Shared configuration files

The standard shared configuration files are very extensive. They are fully documented in the Resources section of the converter JAR file. See *\_read\_me\_first.txt* in that area for details.

# Conversion policy

As will have become apparent by this stage, USX to Sword conversion is a far from straightforward process. As a result it has been necessary to make certain policy decisions. Some of these have been discussed above, but it may be useful to draw them altogether here. The following are in no particular order.

* **OSIS compliance**: Crosswire require that any OSIS files made available to them conform to the OSIS XSD. [[15]](#footnote-15) As explained in section 8.5, we aren’t fully compliant and have no clear way of remedying this. (We do intend to try to make modules available to third parties nonetheless, in the hope that they will still be usable.)
* **Sword compliance**: Corollary to the previous point. It is not entirely clear whether a Sword module produced from non-compliant OSIS is itself acceptable to Crosswire. The fact that it works for us does not necessarily mean it will work in other applications.
* **Cross-boundary markup**: USX permits (and by implication, encourages) markup running across verse boundaries. So does OSIS. And in theory, *osis2mod* can cope as well. The trouble is that in practice, there are certain circumstances where it cannot. To try to avoid issues, I am making three general changes to the raw USX:
* I reserve the option of replacing enclosing para:p by self-closing para:p, positioned at the start of the paragraph. The acceptability of this, however, depends upon the fact that a para is formatted purely by means of vertical whitespace at the start, rather than using an indented first line or having vertical whitespace at the end – or indeed most other things you might think of.[[16]](#footnote-16)
* I am making very significant changes to tables, discussed in section 10.2.
* As discussed above, I am suppressing OSIS list-bracketing tags currently.
* **Tag conversion**: As far as possible, I have followed Appendix F of the OSIS reference manual, which contains SIL’s recommendations for mapping USX tags to OSIS tags. However, given that in many cases there really *is* no obvious mapping, this is at best somewhat *ad hoc*, and I have departed from this wherever it seemed expedient to do so. Most mappings are actually configurable on a per-text basis[[17]](#footnote-17) (you just need to override definitions in *usxToOsisTagConversionsEtc.conf* in the JAR’s *Resources* section), so that you could, for instance, map *char:add* to one OSIS construct for one text, and to another construct for another. I would recommend reconfiguring them only *in extremis*, though.
* **Cross-references**: Cross-references are complicated, in that both USX and OSIS seem to have two different ways of representing them (although needless to say, there is no direct mapping between the two).

USX *ref* tags are generally easy. The only real complication comes where reversification changes verse numbering. In this case, I update the *loc* parameter (which is in USX format) to point to the right place. At one stage I also felt the need to update the content of the *ref* tag content (which gives the same reference in vernacular form). This, however, would have been difficult, because it requires a knowledge of the syntax of vernacular references; and fortunately I have since been persuaded by DIB that there is no need to do so because of the way we are marking up reversified text.

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, as mentioned above, *ref* cannot.[[18]](#footnote-18) 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 will be difficult.

**\* End of document \***

1. The code base does actually include a class which converts from Verse-per-Line (VL) format to USX. However, it appears that there is some disagreement as to precisely what VL is, so this particular code is almost certainly going to require tailoring to each specific text. [↑](#footnote-ref-1)
2. In fact, we now also have a proprietary version of this tool to support reversification and also to cater for non-standard Bibles, where perhaps the translators have chosen to include the verses in a non-standard order. We use both this proprietary version and the Crosswire version, depending upon what we are handling and what we are attempting to achieve. [↑](#footnote-ref-2)
3. According to my notes, our DLL is the same as the Crosswire one, but with additional functionality. It should therefore be possible to use ours in place of the Crosswire one if you choose to. I’ve never quite had the courage to try it. [↑](#footnote-ref-3)
4. If you are working with DBL texts, this will usually appear in the DBL *metadata.xml* file and you can pick it up from there. [↑](#footnote-ref-4)
5. Information is built into the *Resources* section of the JAR file which lets the processing extract elements from DBL metadata and transfer it into the converter’s internal data structures, thus bypassing the need for you to transcribe this information manually into the comfiguration data. Be aware, though, that the DBL metadata is quite complicated, and I’m not entirely convinced it is always used consistently, so relying upon this built-in processing may not always work. [↑](#footnote-ref-5)
6. If you need a semicolon *within* one of the settings, give it as \;. If you need a backslash, give it as \\. [↑](#footnote-ref-6)
7. *osis2mod* is supplied by Crosswire. Because this document is essentially describing the Windows environment, the version of *osis2mod* of interest here is the Windows version. You should be aware that in fact Crosswire guarantee to maintain only the Linux version of the utility, although in practice the Windows version is being kept reasonably up to date at the present time. Latterly we have been experimenting with our own version of osis2mod to help with things like reversification, although we will still continue using the Crosswire version where we are producing modules which we intend to make publicly available. [↑](#footnote-ref-7)
8. As mentioned earlier, this is not actually entirely true. We can cope with USFM input indirectly, because UBS’s Paratext tool can convert USFM to USX. And I do have code to handle Verse-per-Line (VL) format input. However, the code for the latter is not properly integrated with the rest of the system because it appears that there is no common view as to precisely what VL is, and therefore the code needs changing on a case-by-case basis. [↑](#footnote-ref-8)
9. Of course you are free to pre-process the text entirely outside the ambit of the converter, and present the converter with ‘corrected’ USX. But as explained in section 6, I also support automated pre-processing. [↑](#footnote-ref-9)
10. SIL have supplied an appendix in the OSIS reference manual which gives proposed mappings, and I have used these as a starting point. However, some of them are rather complicated, and don’t really work for us. [↑](#footnote-ref-10)
11. All of my tags and attributes have names starting ‘\_X\_’. [↑](#footnote-ref-11)
12. As a perhaps not very realistic example of what might be involved, you might have a verse whose start appears within one paragraph, and whose end appears in the markup right at the start of another paragraph – ie the verse runs across the paragraph boundary. In this case, the cross-boundary markup can be avoided if the end marker is moved out of its containing paragraph, and inserted at the end of the paragraph containing the start marker. [↑](#footnote-ref-12)
13. In fact, STEP does have its own style sheet, which I came across once, but have never since managed to find again, and we were able to make a modification to this in order to fix a minor problem with rendering. But to respond to things like the fact that DBL specifies formats on a per text basis would presumably require STEP to accept a different version of its internal style sheet for each text, and there is, so far as I know, no mechanism to achieve this. [↑](#footnote-ref-13)
14. This is an issue to do with the way the text is rendered: it is not an artefact of the conversion process. I believe, in fact, that it may now have been fixed, but have yet to check. [↑](#footnote-ref-14)
15. Bear in mind, though, that even 100% OSIS compliance may not be entirely meaningful. In respect of tables, for instance, the OSIS reference manual admits that its support is at best half-hearted, and suggests that recourse to user defined attributes may be necessary. But as soon as you rely upon a user-defined attribute to make things work, worrying about compliance to OSIS becomes a little pointless, since no one else will be able to make use of your data anyway. [↑](#footnote-ref-15)
16. I tend to blow hot and cold on this particular modification, so there is little point in me saying here whether I am applying it or not – by the time you read this, I may well have changed my mind about what I am doing. [↑](#footnote-ref-16)
17. A few mappings, however, cannot be altered via configuration changes, because they involve actions too complicated to express using configuration parameters, and their processing is therefore hard-coded. [↑](#footnote-ref-17)
18. There seems to be an urban myth that STEP can handle individual cross-references which are themselves in the form of a collection. It can’t. [↑](#footnote-ref-18)