|  |  |
| --- | --- |
| OASIS Baseline Scanpath Source Code  Documentation |  |
|  | |
|  | |

**Project 3024, Acceleration of Large-Scale Additive Manufacturing**



(Released: August 2020)

A picture containing drawing

Description automatically generated

**Submitted by:**

David Toledano  
Systems Engineer

GE Research

1 Research Circle

Niskayuna, New York 12309

Phone: 518-387-5189  
Fax: 518-387-7080

**ncdmm.org**

**americamakes.us**

"This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8650-16-2-5700. The U.S. Government is authorized to reproduce and distribute reprints for. Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government."

List of Figures 3

List of Tables 3

1. Executive Summary 4

2. Introduction and Problem Statement 4

2.1. Use case 4

2.2. Features of the ALSAM XML scan schema 4

2.3. Limitations 5

2.3.1. Baseline code limitations 5

2.3.2. Potential machine limitations 6

3. End-user requirements 6

3.1. Pre-compiled code execution 6

3.2. Source code compilation 6

4. The Visual Studio Solution 7

4.1. Solution overview 7

4.2. Executable process 7

4.3. XML creation via the Microsoft Domain Object Model (DOM) 8

4.4. SVG visualization via simple\_svg\_1.0.0.hpp 8

4.5. Shared\_files folder 8

4.5.1. readExcelConfig and BasicExcel 8

4.5.2. errorChecks 9

4.5.3. constants 9

5. Project createScanpaths 9

5.1. Output folder structure set up by createScanpaths 9

5.2. Process outline of main\_createScanpaths.cpp 10

6. Project genLayer 11

6.1. genLayer inputs 11

6.2. genLayer outputs 11

6.3. The XML Layer format 12

6.4. Slic3r 12

6.5. Process outline of main\_genLayer.cpp 13

7. Project genScan 14

7.1. genScan inputs 14

7.2. genScan outputs 14

7.3. XML scan file content 15

7.3.1. Header 15

7.3.2. VelocityProfileList 15

7.3.3. SegmentStyleList 15

7.3.4. TrajectoryList 16

7.4. Mapping configuration-file “regions” to trajectories and paths 17

7.4.1. Grouping parts and regions into trajectories 17

7.4.2. An example 17

7.5. Offsetting (indenting) parts via Clipper 18

7.6. Hatching algorithm 19

7.7. Process outline of main\_genScan.cpp 20

List of Abbreviations 21

List of XML Scan Schema Terminology 23

List of Figures

Figure 1 Visual Studio Solution structure 7

Figure 2 Baseline code execution process 8

Figure 3 Output folder structure set up by createScanpaths.exe 10

List of Tables

Table 1 Sample Parts table, for trajectory generation example 18

Table 2 Sample RegionProfile table, for trajectory generation example 18

Table 3 Results of trajectory generation example 18

**Disclaimer**

Unless otherwise noted, modules developed by GE under ALSAM project 3024 are Copyright 2020 America Makes and are licensed under the BSD 2-clause license. The following text appears at the start of each file:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Executive Summary

This document describes the C++ source code included with ALSAM project 3024 to generate scanpaths for powder-bed fusion Additive Manufacturing (PBFAM) involving lasers and metallic powders. The code was developed as a baseline method to create sets of hatches and contours in an XML format which may be fed to an open-source machine controller. The XML format is intended for use on a wide range of PBFAM machines, pending the development of a compatible machine controller for any particular class of machines

The baseline code is capable of generating all current features of the XML schema as described in “ALSAM3024 multiLaser XML schema 2020323.docx” although it has numerous limitations (and opportunities for expansion) as noted in following sections

End-user requirements for executing the compiled code are included in this document, but instructions on features and configuration are provided in a separate document

# Introduction and Problem Statement

## Use case

This code is intended to generate build files which demonstrate features of the ALSAM XML scan schema. Parts are to be defined via STL files. Individual builds are be described via an Excel-based configuration file which indicates the filename and build position of each part, assigns scan strategies to hatches and/or contours by part, and indicates the relative build order of each part

Within an individual part, the only scan-strategy differentiation provided by this code is between hatches and contours (i.e. all hatches of part X receive the same scan strategy, irrespective of geometry)

As an intermediate step prior to generating scan files, the code generates XML layer files, using an ALSAM XML layer schema. This simplifies the overall build process by separating part-slicing from scanpath creation and allows alternate scanpaths to be regenerated from the same set of layer files. However, use and creation of XML layer files are not end-user requirements because layer files are not utilized by the machine controller code

The code was developed for use on a Windows machine and reads a Microsoft Excel-based configuration file. However, the XML scan schema itself is intended to be platform independent

## Features of the ALSAM XML scan schema

See “ALSAM3024 multiLaser XML schema 2020323.docx” for details of the schema itself. To recap the features of the scan schema itself,

* The XML scan schema provides a machine-independent method of describing build files for powder-bed fusion additive machines. The scan files themselves are machine-independent, analogous to PDF-formatted documents
* The schema provides fields to describe and configure various scan strategies (combinations of laser power, velocity and other factors) and apply these strategies to build trajectories (segments of marks and jumps). A separate XML scan file is to be generated for each build layer, named in numerical order, for example scan\_001.xml, scan\_002.xml … scan\_135.xml (with sufficient leading zeroes in the layer number to ensure proper alphanumerical sorting)
* Scan files are to be interpreted by a Machine Controller to be developed by end users for their particular model of printer. Under the current ALSAM project, a Machine Controller is provided for a GE Concept Laser M2 machine
* Within the schema, scan strategy and coordinates can be altered segment-by-segment to enable complete control over difficult geometries
* The schema does *not* support dynamic scan strategies, in which scan parameters within an individual layer are adjusted based on real-time sensor feedback from the same layer. However, such feedback could be used to alter the parameters used to generate build files for *successive* layers. This is not a feature of the baseline source code

## Limitations

### Baseline code limitations

The scan schema provides much more flexibility than the ALSAM 3024 scanpath generation code and configuration file. Some limitations of the baseline code, which could be addressed by third-party development, include:

* Layer thickness is constant across all layers of the build, based on the value indicated in the configuration file
* All parts in the build are sliced at this same layer height, and all parts are hatched/contoured on all layers. Skipping layers, to achieve higher layer heights for individual parts, is not implemented
* Parts cannot be subdivided into smaller areas to apply individual scanpath strategies
  + Each part is permitted a single strategy (a combination of laser power, velocity and other parameters) to be applied to all its hatches, and a separate strategy for all its contours – across all layers and regions
  + The following values are also held fixed across all layers of an individual part: hatch/contour offsets, hatch-to-hatch spacing and contour-to-contour spacing
  + If a part is replicated in the same build, however, each copy can be assigned individual pairs of hatch/contour strategies
* The fixed sets of scan strategies per by part are applied without respect to local geometry, x/y/z thicknesses, layer height, over/underhangs or other conditions
* The same laser is used for all hatches of a particular part on all layers, irrespective of part size
* All build files are generated prior to start of the build. During the build, XML files for upcoming layers are not regenerated or altered based on sensor feedback from current and prior layers

### Potential machine limitations

The scanpath code conducts basic (“sanity check”) analyses on the configuration file to highlight infeasible conditions. Such conditions include referencing a nonexistent (undefined) scan strategy, calling for an STL file which cannot be found, or specifying a zero or negative scan velocity

However, since the code is intended to be machine-independent, it places few restrictions on the range of machine parameters specified in a configuration file. Such “unrestricted” parameters include the number of available lasers, min/max layer thickness, maximum laser power and scan velocity, and maximum build dimensions (x/y range and z-height). End-users must ensure that their XML build files comply with their specific machine limitations

Laser ID’s (the tag used in XML scan files to indicate use of a specific laser on a particular segment) are intended to be semi machine-independent via the use of an ID-to-serial-number lookup file utilized by the machine controller. Thus, the build configuration file should typically refer to lasers by generic ID’s such as 1, 2, 3 or 4

# End-user requirements

## Pre-compiled code execution

Requires:

* A Windows environment
  + This is a requirement of the compiled executables, which perform numerous system calls for file navigation and other functions, rather than a restriction of the XML scan schema or machine controller
  + The executables have not been tested under non-Windows environments
* Copies of the three executables (createScanpaths.exe, genLayer.exe, genScan.exe), placed in a common folder. These may be downloaded from the PBFAM or OASIS GitHub sites via the Multilaser\_Pre-compiledBinaries repository or zip file
* Download of the slic3r 1.3.0 package from www.slic3r.org and unpacking its contents into the slic3r folder included in the precompiled-binaries executable folder
  + This package is not included in source code due to its license details and size (it’s considerably larger than the ALSAM scanpath package)
* An Excel configuration file describing the build
  + This file has a specific format and version number. Several examples and a blank file are included in the precompiled-binaries repository
  + Excel 97-03 (.xls) format is used, because this format is compatible with open-source editors such as Apache OpenOffice

## Source code compilation

* The code was developed and compiled using Microsoft Visual Studio 2015 as Solution, described below. The VS Solution involves several Windows-specific packages, including windows.h and msxml6.h
* Porting to a different IDE and compiler has not been evaluated. However, the existing code should be compatible with the free Microsoft Visual Studio Community edition

# The Visual Studio Solution

## Solution overview

The Visual Studio Solution is divided into three Projects (in separate folders) which each produce an executable file: createScanpaths, genLayer and genScan. End-users interact only with createScanpaths.exe, which provides file navigation, basic error checking and coordination of layer and scan generation. The other two executables, genLayer.exe and genScan.exe, are called by createScanpaths to generate layer and scan files, respectively

The Solution contains an additional folder (shared\_files) which contains a number of source-code modules used by all three Projects

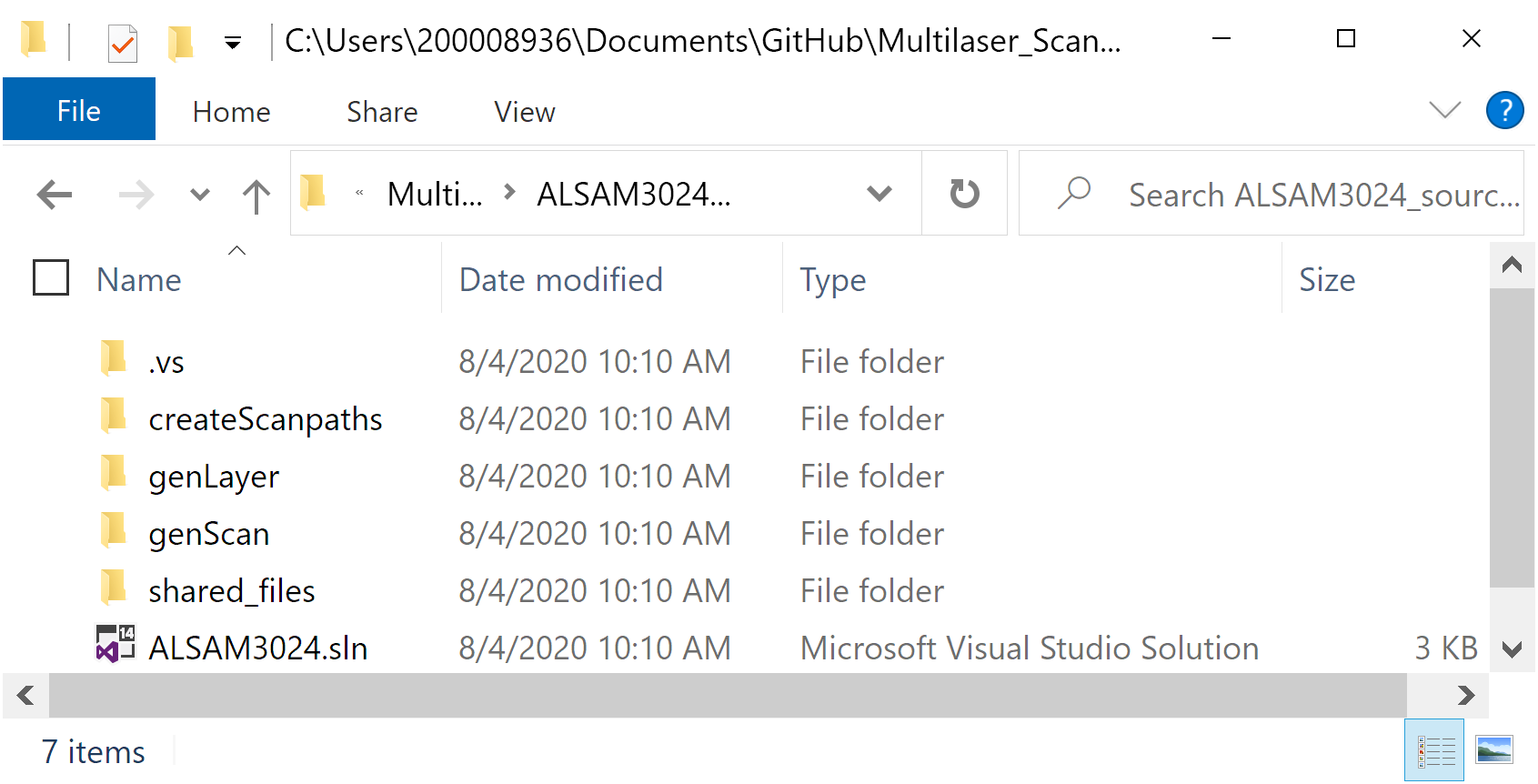


Figure 1 Visual Studio Solution structure

## Executable process

Figure 2 outlines the process implemented by the baseline code and the three executables. Prior to executing createScanpaths.exe, end-users must set up an Excel configuration file which provides a project name, describes the part (STL) files and defines scan strategies to be applied to each part region (hatch or contour)

createScanpaths.exe will create a new folder in the same location as the selected configuration file, using the project name given in the configuration file. The new folder will contain subfolders for layer and scan files

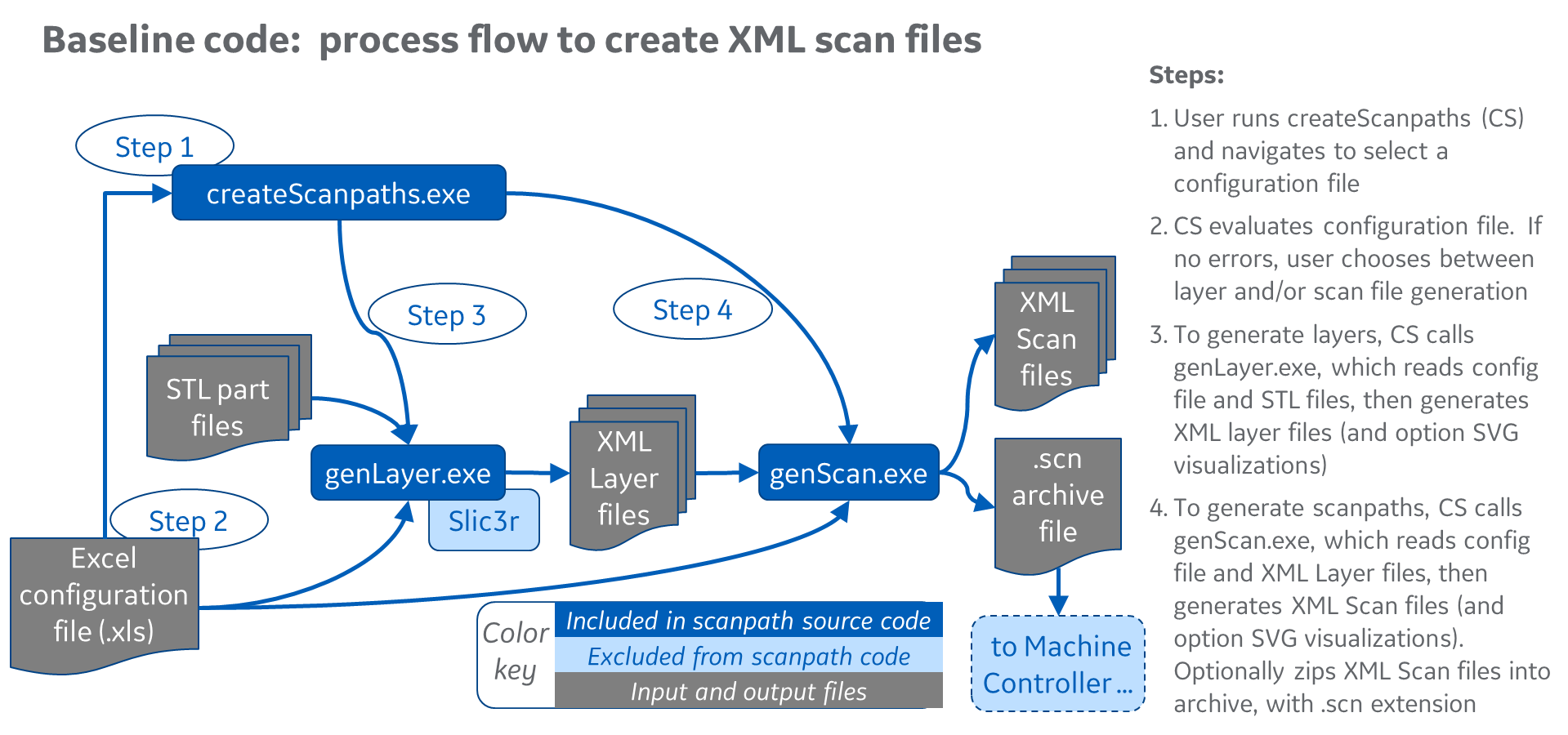


Figure 2 Baseline code execution process

## XML creation via the Microsoft Domain Object Model (DOM)

The DOM, via msxml6.h, is used by genLayer and genScan to parse fields into an XML format. DOM allows an XML tree to be accumulated, provides XML tag management and enables the tree to be written to a text file [via save() at the end of writeLayer()]. writeLayer() in writeLayerXML.cpp demonstrates how the package may be used

This package permits only a single XML tree to be generated at one time. The CoInitialize() command in main\_genLayer.cpp clears and instantiates the tree and CoUninitialize() destroys it. The actual tree “variable” is not exposed, and any nodes added are added to this same tree

main\_genLayer.cpp uses CoInitialize/CoUninitialize twice: once to create and save (to text file) the XML layer structure, and a second time to create and save a small XML header file

## SVG visualization via simple\_svg\_1.0.0.hpp

This package contains commands to generate lines in SVG format and is used by genLayer and genScan to create optional visualization files for some or all layers. genLayer displays outer contours in black and inner contours in blue. genScan displays all “mark” segments (with power>0) as black (contours and hatches) and does not display jump segments

## Shared\_files folder

Code modules found in the shared\_files folder within the Visual Studio solution are used by all three Projects. Significant modules include the following:

### readExcelConfig and BasicExcel

readExcelConfig parses the tabs and fields of a configuration file in Excel 97-2003-format. Its primary function, AMconfigRead(), accepts a file name with path and returns a structure (AMconfig) containing all configuration information

The expected names of the tabs in the config file are hard-coded in readExcelConfig.h, and the field locations and formats are hard-coded in AMconfigRead()

BasicExcel (.cpp, .hpp) is an independent package used to read Excel files in the ’97-2003 format. No file creation or editing functions are used. Known limitations of this package include:

> Crashes if any field (including comments/notes) exceed 255 characters, but often only on the second time the file is opened by createScanpaths

> Does not properly read cells containing formulas. Interprets these as text cells and may return a formula rather than the cell value

### errorChecks

Provides a basic error-handling routine (via updateErrorResults) and performs extensive “sanity checks” on the configuration file (via evaluateConfigFile)

evaluateConfigFile() is only called by createScanpaths when the configuration file is selected by the user. genLayer and genScan assume that any config filename passed to them has passed the checks

updateErrorResults() accumulates error messages into an errorCheckStructure. Depending on the parameters sent to the function it will either accumulate and continue; or, it will write the most-recent message to console, write all accumulated messages to a text file, and halt execution of the entire scanpath-generation process

### constants

Contains a number of critical elements including the compatible configuration-file version (which any config-file must meet or be cancelled), output XML schema version, coordinate precision and other factors

# Project createScanpaths

This executable manages configuration file selection, folder setup, configuration error checking and coordination of layer and scan generation but does not create output directly. createScanpaths requires an Excel ’97-2003 configuration file and the executables genLayer.exe and genScan.exe. Configuration files may appear in any folder that the user can navigate to (and has rights to create an output folder), however, the executables are expected to appear in the same folder as createScanpaths.exe

## Output folder structure set up by createScanpaths

After reading the configuration file, createScanpaths creates an output folder structure (the “project folder”) in the directory that contains the configuration file. The project folder is named per a field on tab 2 of the config file. The output folder structure is shown in Figure 3

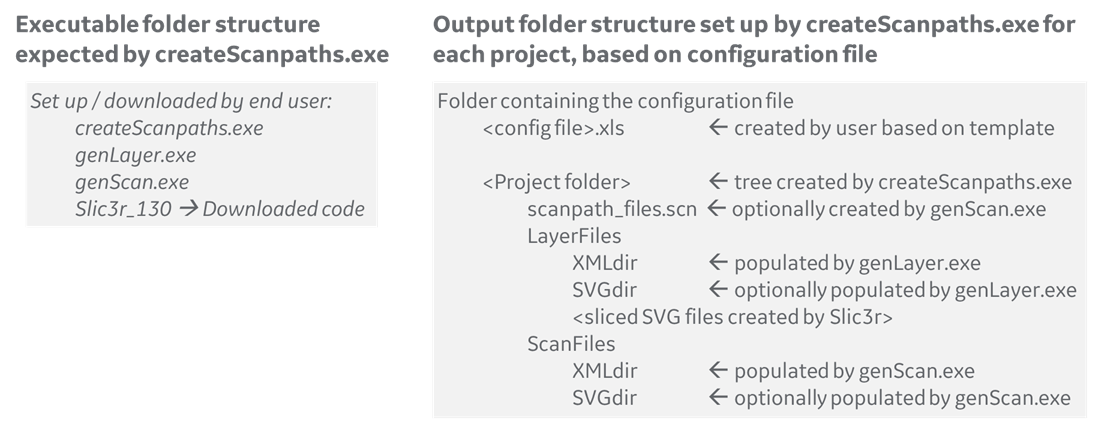


Figure 3 Output folder structure set up by createScanpaths.exe

## Process outline of main\_createScanpaths.cpp

1. Clean up any existing status or error files from prior runs
2. Ask user to navigate to and select a configuration file via selectConfigFile(), which uses Windows command GetOpenFileNameA
3. Read the configuration file via AMconfigRead(), then check it for errors via evaluateConfigFile(). If errors are found, report the error to console and a text file and quit
4. Otherwise, evaluate the project (output) folder listed in the configuration file via evaluateProjectFolder() to see if there are existing scan and/or layer files. This determines the options available to the user
5. Ask user which of the available options to execute via getUserOption(). May include generating only layer files, only scan files (if layer files already exist), or both. Then ask whether to overwrite or merge with existing files
6. Set up output folder structure and/or delete existing results via setupOutputFolders()
7. If layer-file generation is selected, set up a loop in callGenerationCode() to execute genLayer repeatedly until all layer files are generated

* Due to a persistent memory leak, we generate a limited number of layers via each call to genLayer. Currently, genLayer creates 25 and then quits
* genLayer reports its status through both its return code (indicating whether an error occurred) and a text file named gl\_sts.cfg (which indicates the last layer generated and whether all layers are complete)
* callGenerationCode() polls gl\_sts.cfg and returns upon error or completion of all layers

1. If scan-file generation is selected, set up a loop in callGenerationCode () to execute genScan repeatedly until all scan files are generated

* As in the previous step, we generate a limited number of layers via each call to genScan (also 25)
* Also, as in the previous step, genScan reports its status through both its return code (indicating whether an error occurred) and a text file named gs\_sts.cfg (different status file from that used by genLayer)
* callGenerationCode () polls gs\_sts.cfg and returns upon error or completion of all layers

1. Following successful scan file generation, the files are optionally packaged into a zip-style archive by createScanZipFile(). This uses external zip.cpp package which creates and populates zip archives. The corresponding unzip code (unzip.cpp) is not included in this solution
2. Clean up by deleting any status files and Slic3r output (SVG) files that remain in the executable folder

# Project genLayer

genLayer.exe is called by createScanpaths to read parts in STL format and use information from the configuration file to generate XML layer files as an intermediate step in XML scan file creation. The external Slic3r package is used to “slice” each STL file into uniform-thickness layer, which Slic3r outputs as an SVG file (containing contour outlines for all layers of a particular STL file). genLayer then reads these layer SVG files, assigns tags to each contour to associate it with a part listed in the configuration file, and combines all parts into individual layer XML files

This module processes a limited number of layers at one time, due to small memory leaks. Currently this is set at 25 layers per call to genLayer.exe. The module generates either the next 25 layers or the total remaining, if less than 25, and writes its final layer number and whether all layers are completed to status file gl\_sts.cfg

## genLayer inputs

Excel configuration filename, passed as a command-line parameter from createScanpaths. The full file path is expected, enclosed in double quotes. genLayer will re-read the config file each time it is called

One or more STL files, each containing one or parts. Either text or binary STL format is acceptable. STL files must be located in the same folder as the configuration file

The LayerFiles output file structure described in Figure 3 must be set up by createScanpaths prior to calling genLayer

## genLayer outputs

One or more layer XML files in <Project folder>\LayerFiles\XMLdir, named layer\_xxxxx.xml, where xxxxx represents the integer layer number starting from one. See format below

Optionally, one or more SVG visualization files in <Project folder>\LayerFiles\SVGdir, named layer\_xxxxx.svg. These files provide a quick means of visualizing the contours generated for a particular layer and may be opened in most web browsers

Slic3r output: the SVG contour files created by Slic3r for each part will be moved to the output LayerFiles folder

Status file: gl\_sts.cfg, a four-line text file, created or modified to communicate status with createScanpaths. Indicates the final layer generated in the last run of genLayer.exe and whether all layers are completed. This file is normally deleted by createScanpaths following completion of all calls to genLayer

## The XML Layer format

Sample layer files may be found in the precompiled binaries folder under any of the examples, in <ProjectFolder>\LayerFiles\XMLdir

Each XML file covers a single layer and consists of three top-level fields: Thickness (a single field), VertexList and Slice

*Thickness* indicates the z-thickness of the layer, in mm

*VertexList* lists all vertices (intersection points) on all contours

*Slice* contains one or more Regions, which are closed contours identified with a specific part file. The layer file contains a single slice which lists regions from all parts present on the layer. An individual part, if it appears on a particular layer, may have any number of regions based on the inner/outer contour content of that height in its STL file

Each region is defined by the following fields:

* Type: Inner or Outer, indicating whether the region encloses actual part (Outer) or an interstitial space (Inner)
* Tag: Alphanumeric value that identifies the region profile assigned to this part in the configuration file. Region profiles define hatch and contour scan strategies, offsets, spacings and other parameters. Note that the tag does not necessarily link the region to a specific part, since multiple parts may share the same region tag (i.e. you may not be able to tell which part generated a particular region)
* contourTraj (contour trajectory): Integer value that indicates the build order and grouping of any contours that will be built from this region. Traj = Trajectory, which is a grouping of scanpaths. Trajectories will be built in numerical order, lowest first. If multiple regions (hatches and/or contours from one or multiple parts) have the same contourTraj number, they will be built as a group. If no contour trajectory value is assigned to a part in the config file, its regions will receive contourTraj = 9998
* hatchTraj (hatch trajectory): Integer value indicating the build order and grouping of any hatches that will be built from this region. Interpretation is analogous to hatchTraj, above. The configuration file permits separate trajectory numbers to the contours and hatches of each part. If no hatch trajectory value is assigned to a part in the config file, its regions will receive hatchTraj = 9999

Following the four fields above, each Region contains a list of Edges (line segments) which are defined in terms of Start/End vertex numbers. The vertex numbers don’t appear in the layer file; they are determined by counting

## Slic3r

The Slic3r package ([www.slic3r.org](http://www.slic3r.org)) is used to slice each STL file into one or more layers. Source code is not included in this solution; end-users are instructed to download the executable and unpack it in a specific folder

genLayer accesses Slic3r once for each STL file via a call to slic3r.exe containing the name of the STL file. Slic3r does not provide an error return value if it runs into a problem, so we simply check for the existence of the appropriately-named SVG output file. See runSlic3r() for details

## Process outline of main\_genLayer.cpp

The numbering below corresponds to comment numbering in this file

1. Read the configuration file indicated by command-line argument in the call to genLayer.exe
2. Read the status file (gl\_sts.cfg) to determine the last layer completed by prior run of genLayer.exe, if any. Use this to set the number of layers to be completed in this genLayer instance and determine whether this will complete all layers of the build
3. Iterate through the list of STL part filenames to slice them into layers (if an SVG file of the same name does not already exist) and/or compute x/y range per part from the Slic3r-generated SVG file

3a. If this filename hasn’t been encountered yet in the file list (will only be true if the same part is duplicated in the build), verify that the STL file exists. Quit with error if not found

3b. Determine bounding box of the part, incorporating offsets. If it hasn't been sliced, compute from STL file via findBoundary(), otherwise pull from prior data. Store in vv and vs

3c. If not previously sliced, run Slic3r on the part via runSlic3r()

3d. Compute number of layers in this part above z=0, incorporating z offset

1. Determine maximum number of layers in the entire build. This is the greatest layer height of any part, as well as layer heights of any single stripes specified in the configuration file (which will not appear in layer files since they have no STL outlines). Single stripes are evaluated here because otherwise we might not create any layer files for 1-2 layer “bead on plate” experiments, which would preclude creating any scan files (scan files are only created for corresponding layer files)
2. Generate scaling factors for SVG visualization and save to a text file to be later used by scan generator
3. Iterate through the target set of layers for this instance of genLayer (from sLayer [starting layer#] to fLayer [ending layer#]). Interval is typically 25 layers except when fewer than 25 layers remain to be processed

6a. Iterate across parts listed in vOBJ, generate a layer structure for each and append to vLayer, via readFile() and scaleLayer(). “layer” is a structure which identifies a height and thickness and contains a list of vertices (vList) and two potential slices (upper & lower). Currently we only populate and reference the upper slice, layer.us. This step is essentially reformatting the SVG contours of each part and adding appropriate region tags and trajectory numbers as indicated in the configuration file

6b. Combine all parts into one layer structure, via combLayer()

6c. Clean up the layer structure and finalize the format, via refineLayer()

6d. Optionally, generate an SVG visualization file displaying just this layer, via rlayer2SVG()

6e. Generate an XML layer file from the layer structure, using the Microsoft DOM (Domain Object Model), via writeLayer()

1. The target number of layers are now generated for this instance of genLayer. Create a single XML file containing header information from the DOM
2. Write ending layer number and whether all are completed to gl\_sts.cfg file for communication with createScanpaths

# Project genScan

genScan.exe is called by createScanpaths.exe to read individual layer files in XML layer format and use information from the configuration file to generate hatches and contours. genScan creates an XML scan file for each layer which lists mark and jump segments (grouped and ordered into Trajectories and Paths, described below) preceded by a list of all scan and jump strategies utilized in the build (the SegmentStyleList and VelocityProfileList sections). All segments, including jumps, are assigned a SegmentStyle

As with genLayer, this module processes a limited number of layers at one time, due to small memory leaks. Currently this is set at 25 layers per call to genLayer.exe. The module generates either the next 25 layers or the total remaining, if less than 25, and writes its final layer number and whether all layers are completed to status file gs\_sts.cfg (named differently from the file used by genLayer)

## genScan inputs

Excel configuration filename, passed as a command-line parameter from createScanpaths. The full file path is expected, enclosed in double quotes. genScan will re-read the config file each time it is called

One or more XML layer files located in <project folder>\LayerFiles\XMLdir. Scan files will be created for corresponding layer files

The ScanFiles output file structure described in Figure 3 must be set up by createScanpaths prior to calling genLayer

## genScan outputs

One or more scan XML files in <Project folder>\ScanFiles\XMLdir, named scan\_xxxxx.xml, where xxxxx represents the integer layer number starting from one

Optionally, one or more SVG visualization files in <Project folder>\ScanFiles\SVGdir, named scan\_xxxxx.svg. These files provide a quick means of visualizing the mark segments generated for a particular layer and may be opened in most web browsers

Status file: gs\_sts.cfg, a four-line text file, created or modified to communicate status with createScanpaths. Indicates the final layer generated in the last run of genScan.exe and whether all layers are completed. This file is normally deleted by createScanpaths following completion of all calls to genScan

## XML scan file content

The XML scan format is fully described in “ALSAM3024 multiLaser XML schema 2020323.docx.” The following sections briefly note the major top-level areas of the schema, and how these items are defined in the baseline scanpath code. Care has been taken to differentiate schema requirements from baseline code implementation, but if in doubt about actual schema capabilities, please refer to the schema document above

### Header

This first section of the schema contains a few elements that help identify the file and its position in the build

AmericaMakesSchemaVersion is currently 2020-03-23 (last update point)

LayerNum should match the number in the filename, but it typically not evaluated

LayerThickness (in mm) may be varied from layer-to-layer, although the baseline code uses a constant thickness as indicated in the configuration file

AbsoluteHeight may be useful in cases where LayerThickness varies. The baseline code, however, computes this simply as LayerNum \* LayerThickness

DosingFactor is taken directly from the configuration file

BuildDescription is intended as free text, although it may be useful to list min/max ranges of various parameters used in the build file, such as laser power

### VelocityProfileList

A VelocityProfile (VP) in the XML scan schema defines a specific combination of laser velocity and delays. All such combinations used in the layer must be listed and defined in the VelocityProfileList section which precedes the list of segment. A VP combined with other laser parameters defines a SegmentStyle, as described in the next section

The list of VP’s which genScan outputs to the build file will be identical to the list defined by the user on the VelocityProfiles tab of the configuration file. No VP’s will be omitted, even if they are not used in the entire build or a particular layer

Any number of VelocityProfiles may be created, to permit very granular control of build parameters. Each VP may be reused in multiple SegmentStyles. If a SegmentStyle calls for a VP which does not exist, this will cause an error in either scanpath generation or machine execution. However, it is fine to define VP’s which are not used

### SegmentStyleList

Each line segment in the scan file, both mark and jump, must be assigned a SegmentStyle (SS). A SegmentStyle defines a combination of laser ID, laser power, velocity (via reference to a VelocityProfile) and other parameters

All SegmentStyles used in a particular layer must be defined in the SegmentStyleList section of the XML scan file. In creating this list, genScan will start by including all user-defined styles on the SegmentStyles tab of the configuration file. To this list, genScan will add auto-generated “jump” SegmentStyles corresponding to the jump VelocityProfile indicated for each RegionProfile listed in the configuration file. This is simply a convenience so that the user does not need to create both a VelocityProfile and SegmentStyle for jumps; only a VelocityProfile

Regarding jump segments: SegmentStyles intended for jumps (power=0) can be simplified by omitting the laser power parameter; this will be interpreted as zero power. Further, if the laser ID parameter is omitted, it will be assumed that the laser used in the immediately-preceding segment is to perform the jump (per build file order, in the same path). In this way, as few as one generic “jump” SegmentStyle may be used in combination with any number of multi-laser mark styles

Finally, it should be clear that SegmentStyles (by themselves) do not indicate geometric parameters such as hatch spacing or offset. Geometric parameters are implemented via the relative positioning of different segments, whereas SegmentStyles simply indicate what to do when encountering a particular type of line segment

### TrajectoryList

The final major section of the XML scan file contains the actual line segments, grouped into trajectories and paths

A trajectory defines one or more paths (groups of related segments) which must be fully completed before the next trajectory can be started. Trajectories will be built in the order they appear in the XML scan file, irrespective of TrajectoryID (but note that genScan uses trajectory numbers to determine how to order things when generating the XML)

Each trajectory contains a number of key fields

* TrajectoryID is for reference and does not affect the build
* PathProcessingMode defines how the various paths within the trajectory are to be built, sequentially (in appearance order) or in parallel, based on the trajectory’s PathProcessingMode parameter. Parallel building will only be realized if there is more than one path, and if the segments of these paths utilize different lasers (via their SegmentStyles); otherwise, the paths will be built sequentially

Each path within the trajectory contains the following items:

* Type is either contour, hatch or single\_stripes (note that the baseline code separates contours and hatches from each part into separate paths, although they may still be in the same trajectory)
* Tag corresponds to the RegionProfile (from the config file) which was used to select parameters for all segments in the path. A RegionProfile defines both hatch and contour parameters, so the baseline code will assign the same tag for contours and hatches of a particular part, even though the actual SegmentStyles and geometric values may differ between contour and hatch paths
* SkyWriting mode is taken from the configuration file as 0, 1, 2 or 3
* NumSegments indicates the total segments in the path. Miscounting will result in an error upon build
* Start indicates the starting coordinate of the first vertex
* Segment defines the SegmentStyle (abbreviated SegStyle) and endpoint of each segment (which is also the starting point of the following segment). Note that segments are defined in a “point to point” format which mimics a continuous contour; if the path is actually to be a combination of alternating marks and jumps, the SegStyle’s must simply be alternated

## Mapping configuration-file “regions” to trajectories and paths

The Excel configuration file utilized by the baseline scanpath code includes a level of abstraction compared to the XML scan schema, because this code does not implement the full segment-by-segment parameter flexibility provided by the schema. In particular, the scanpath code (via the configuration file) assigns a single RegionProfile to each part, which defines a fixed set of contour and hatch spacings and parameters to be used across the part

The schema is agnostic to parts and regions, and is organized as:

TrajectoryList🡪Trajectory(s)🡪Path(s)🡪combinations of (Segment, SegmentStyle)

In contrast, the baseline code implements a flow-down from a part (which has no analog in the XML schema), to a region profile (which also has no analog in the schema, due to its complete flexibility), and ending in paths and segments, as:

Part P🡪RegionProfile R, hatch trajectory number h, contour trajectory number c

RegionProfile R🡪hatch and contour parameters

Hatch path🡪region tag R, segments all using SegmentStyle R[hatches]

Contour path🡪region tag R, segments all using SegmentStyle R[contours]

### Grouping parts and regions into trajectories

To generate trajectories, the baseline code groups together all paths with the same trajectory value. In the example above, contours and hatches for part P might appear in the same or different trajectory depending on the user-assigned contour and hatch trajectory values. Following identification of all trajectory values, genScan cross-indexes all regions (i.e. contours or hatches of specific part) by trajectory number

### An example

As an example, suppose the configuration file defines parts as shown in Table 1, which is a simplified view of the Parts tab including region profile and trajectory assignments

|  |  |  |  |
| --- | --- | --- | --- |
| **Part (filename)** | **RegionProfile tag** | **Contour trajectory** | **Hatch trajectory** |
| Part 1.stl | RP1 | 3 | 3 |
| Part 2.stl | RP2 | 3 | 2 |
| Part 3.stl | RP1 | 3 | 1 |

Table 1 Sample Parts table, for trajectory generation example

And suppose the configuration file defines RegionProfiles as shown in Table 2. Note that hatch parameters are blank for profile RP2, meaning that any region using this tag is should not be hatched

|  |  |  |  |
| --- | --- | --- | --- |
| **RegionProfile** | **Jump velocity profile** | **Contour params** | **Hatch params** |
| RP1 | VP1 | <parameters> | <parameters> |
| RP2 | VP2 | <parameters> |  |

Table 2 Sample RegionProfile table, for trajectory generation example

Combining information from Tables 1 and 2, the baseline scanpath code will identify a total of two “active” trajectories and create trajectories and paths as shown in Table 3. Trajectory 2 does not appear because its only region would be hatches for profile RP2, which are omitted. In addition, regions within a trajectory are grouped into paths based on equivalent types (contour or hatch) and RegionProfile tag values

|  |  |
| --- | --- |
| **TrajectoryID** | **Paths and tags within the trajectory, in order they will appear in this trajectory** |
| 1 | Path 1 = {Part 3 hatches}, tag RP1 |
| 3 | Path 1 = {Part 1 contours + Part 3 contours}, tag RP1  Path 3 = {Part 1 hatches}, tag RP1  Path 3 = {Part 2 contours}, tag RP2 |

Table 3 Results of trajectory generation example

## Offsetting (indenting) parts via Clipper

In the baseline code, contour and hatch offsets may be used to “shrink” a part, not to expand it. The Clipper package is used to perform such offsets, which are applied such that outer contours contract toward the interior of the part while inner contours (holes) expand toward the outer contour

The ClipperOffset function is called via genScan’s edgeOffset() function. Large offsets on this parts may shrink some or all of the part to zero or subdivide areas in additional regions. All these possibilities are handled gracefully by Clipper, and the baseline hatch() or contour() code expects to receive zero or multiple regions from edgeOffset(). ClipperOffset provides options for rounded, mitered or square corners, but the square option is currently hard-coded in edgeOffset()

## Hatching algorithm

The baseline code’s hatch() and hatchOPT() functions use the same basic hatching method. hatch() overlays hatch lines across the entire build area without respect to geometry and may result in large jumps between widely-dispersed parts. hatchOPT() begins with this same set of hatches but then tries to optimize their order to minimize the jump lengths between them

In both algorithms, sets of hatches (mark segments) are determined independently for each trajectory. Within each trajectory, regions with the same RegionProfile value will be hatched together, as shown in the previous section. This iteration per trajectory is carried out in main\_genScan, which calls hatch() for each collection of regions that are that are to be hatched together (same trajectory and region profile tag)

General outline of hatching one or more regions as implemented in hatch(). A region refers to the complete set of inner/outer contours of a specific part

1. Overlay a grid of parallel lines across the area defined by the bounding box of the layer
   1. These lines are drawn based on the current hatch angle, which is computed from the initial angle and per-layer rotation as (initial angle) + (layer number -1)\*(inter-layer rotation angle)
   2. The current angle determines whether the parallel hatch grid progresses (increments) along the X or Y axis. The objective is to avoid issues in which the hatch angle is parallel to an axis, so (to oversimplify) if the hatch angle is “closer” to the X axis (i.e. 0 or 180 degrees) than to the Y axis (i.e. 90 or 270 degrees) we increment along the Y axis, and vice versa
2. Iterate through all regions to apply the indicated hatch offset
   1. Call edgeOffset() with the region’s edges
   2. If any regions survive, add these edges to the list of offset edges
3. Iterate through the set of parallel grid lines
   1. Iterate across all edges in the post-offset list: see if the edge intersects with this grid line. If so, add intersection point to the intersection list
   2. If there are any intersections, eliminate any duplicates (as can happen if the grid line passes through the intersection point of two edges
   3. Sort the intersections in X or Y (based on hatch angle, as described above). This will order them as alternating with outer-contour and inner-contour intersections
   4. Create alternating mark / jump / mark segments. Apply the hatch SegmentStyle to marks and the RegionProfile’s jump SegmentStyle (auto-generated from the indicated jump VelocityProfile) to jumps
   5. Add these segments to the complete segment list
4. If one or more mark segments were created, format it as a single path structure (with appropriate tag and other values, per config file) and return

## Process outline of main\_genScan.cpp

The numbering below corresponds to comment numbering in this file

1. Parse the command-line arguments to identify the configuration filename and path
2. Read the configuration file via AMconfigRead()
3. Determine which layers to process in this function call
   1. Get total layers to process from the Excel configuration file
   2. Read the temporary config file to see which layers have already been done, then specified starting and ending layers, and compare to available layer files in the layer output folder to set ending layer for this genScan instance
4. Load SVG viewer parameters from vConfig.txt, which should have been created by genLayer
5. Generate a list of all region profile tags listed in the config file, to enable indexing from a region's tag to a specific profile in regionProfileList
6. Process the indicated layers for this genScan instance, from sLayer (starting) to fLayer (ending) layer#
   1. Initialize the Domain Object Model (DOM) from msxml6.h; if this fails, we can't create an XML
   2. Read the appropriate XML layer file created by genLayer. This combined all parts into one file
   3. Convert the layer XML data to appropriate data structure in variable layer L via traverseDOM(), then run verifyLayerStructure() to confirm that all region tags that appear in the layer file are listed in the configuration file. (This should only be possible if the configuration file were changed after layer-file generation but before running scan-file generation)
   4. Determine the bounding box of the current layer via getBB(). Will be used to define the x/y span of the hatching grid
   5. Identify the set of trajectory numbers encountered in the layer file, and the set of regions that make up each trajectory
      1. Each region (inner/outer contour) in the layer file has a region tag, hatch trajectory# and contour trajectory#
      2. This step also duplicates each region to create a contour “region” and hatch “region,” since the configuration file’s RegionProfile specifies separate parameters for contours and hatches. If the RegionProfile omits either contours or hatches, the corresponding duplicate region will also be omitted
      3. As indicated above under trajectory processing, regions will be built in trajectory# order (lowest first). Regions with the same trajectory# and tag value will be contoured or hatched together in a common path. Regions with the same trajectory# but different tag value or type (contour/hatch) will appear as separate paths within the trajectory
      4. Note that this step only identifies and populates trajectory and region lists. Scan path creation happens later
   6. Iterate across the trajectory list generated in step (e) in numerical order, and generate scanpaths
      1. Cross-index back to trajectoryList (which contains the actual region data), which is order by appearance in the layer file, not trajectory#. tNum\_position identifies the current trajectory of interest in trajectoryList
      2. Iterate across the regions in trajectoryList[tNum\_position].trajRegions. Group together all regions having the same tag and hatch or contour them together (to do this, we check and set trajectoryList[]. trajRegionIsHatched)
         1. Check whether the region has been hatched (or contoured)
         2. Iterate across all remaining region and aggregate any with the same tag
         3. If the region is a contour, iterate across the indicated number of contours and call contour() for each successive offset, from outer to inner. Add each contour to the trajectory as a separate path (note – this could be improved by merging successive contours into a single path)
         4. If the region is a hatch, call either hatch() or hatchOPT() (as indicated by the RegionProfile) and save the resulting path to the trajectory
   7. Write the XML schema to a DOM and then save the DOM to a file. writeScanXML() proceeds through the XML scan file as outlined in the section above describing this file: Header, VelocityProfileList, SegmentStyleList, TrajectoryList
   8. If user wants to generate SVG files and we are either on the first layer or a multiple of the SVG interval, run scan2SVG()
7. All layers planned for this genScan instance are complete. Write details to gs\_sts.cfg file for createScanpaths

List of Abbreviations

|  |  |
| --- | --- |
| **Abbreviation** | **Definition** |
| ALSAM | Acceleration of Large Scale Additive Manufacturing, an America Makes initiative |
| AM | Additive Manufacturing (or AmericaMakes.us) |
| DMLM | Direct Metal Laser Melting, typically an alternate terminology for PBFAM |
| DOM | Microsoft Domain Object Model, a package for XML tree creation and output via msxml6.h |
| PBFAM | Powder-Bed Fusion Additive Manufacturing |
| STL | Stereolithography file, which portrays a three-dimensional shape in terms of triangular sections |
| SVG | Structured Vector Graphics, a visualization format |
| VS | Microsoft Visual Studio |

List of XML Scan Schema Terminology

This is duplicated from ALSAM3024 multiLaser XML schema 2020323.docx

|  |  |
| --- | --- |
| **Schema term** | **Description and usage** |
| ***AbsoluteHeight*** | Absolute build height of a particular layer. Not evaluated by the code |
| ***AmericaMakes***  ***SchemaVersion*** | XML schema version, which LabVIEW uses to determine the appropriate parser |
| ***BuildDescription*** | Free text which may be used to indicate expected conditions or required capabilities, such as maximum power and velocity occurring in the build |
| ***DosingFactor*** | Indicates the depth of powder to be applied during recoat, as a multiplicative factor of LayerThickness  1.5 = 50% additional dose (1.5 x layer height) |
| ***End*** | Ending coordinate of the current Segment |
| ***Header*** | Collection of parameters which do not fall under the lists of velocity profiles, segment styles or trajectories |
| ***LaserMode*** | Defines the mode of operation when multiple lasers (Traveler sections) are included in a particular SegmentStyle. May be omitted when <2 Traveler sections are included within the style   * Independent = multiple lasers operating on separate build segments. In this mode of operation, PathProcessingMode (below) is referenced for sequencing information * FollowMe = multiple lasers traveling in synchronized fashion along the same set of segments:   + The TravelerID having SyncDelay=0 (required) is the Master; all other (Slave) TravelerID sections must include a SyncDelay > 0 in reference to the Master laser   + Each TravelerID can have separate power, SpotSize and wobble settings   + The indicated VelocityProfileID is used for all lasers, with the Master laser leading |
| ***LayerThickness*** | Indicator of the thickness of this build layer in the system’s defined units |
| ***LayerNum*** | Optional numbering from bottom to top layer. Not evaluated by the code, so may be non-sequential |
| ***Path*** | Set of related scan paths (Segments) which make up a specific aspect of the build. Typically, a part should be divided into multiple Paths such that one path might include all the contours, and another includes all the hatches. All Segments within a particular Path should utilize the same laser(s) to avoid synchronization and timing issues within the Path. Parts can be subdivided into as many Paths as desired |
| ***PathProcessing***  ***Mode*** | Determines how multiple Paths within a particular TrajectoryID will be sequenced   * Sequential (default if omitted): Paths will be processed in the order in which they are listed. Each successive Path will begin building only after its predecessor is completed. This mode assures that the laser(s) used within a particular TrajectoryID are not subject to timing uncertainties (if the same laser(s) are used in multiple Paths) but may result in a longer build * Concurrent: Paths will be built concurrently as permitted by laser availability. This mode will typically be used when each Path is assigned to a different laser. In general, any Paths which utilize unique lasers will be initiated immediately, but timing after any Path is completed will depend on the availability of each laser and cannot be guaranteed |
| ***Power*** | Laser marking power in watts. If the TravelerID section is omitted from a SegmentStyle, Power will be set to 0 and the SegmentStyle is assumed to be a jump |
| ***Segment*** | An individual unit of laser mark or jump between two points. The first Segment in a Path begins at the Path’s Start coordinate and continues to that Segment’s End coordinate (which is also the starting point for the next Segment). The laser(s) which actually carry out the mark or jump are determined by referencing the SegmentStyleID:   * If the SegmentStyle indicates one or more TravelerID’s, these lasers carry out the Segment * If there is no TravelerID listed in the SegmentStyle, the Segment is assumed to be a jump. Power is set to zero and the laser(s) used in the immediately preceding Segment carry out the jump. Therefore, it is improper to begin a Path with a jump segment unless the SegmentStyle specifies a TravelerID to make the jump |
| ***SegmentStyle*** | A set of parameters which defines a single mode of mark or jump operation. Each SegmentStyle must include a VelocityProfileID and may (optionally) include LaserMode and one or more TravelerID (laser parameter) sections:   * Jumps may omit the TravelerID section. In this case the “jumping” laser(s) will assumed to be the laser(s) used in the immediately prior Segment within a particular Path. The laser’s Power will be set to zero during the jump. This may reduce the number of styles needed, since a single jump style can be utilized for all lasers * If LaserMode is omitted or set to Independent, only one TravelerID section should be included in the SegmentStyle * If LaserMode is set to FollowMe (synchronized), at least two TravelerID sections must be included. See LaserMode and TravelerID for further details |
| ***SegmentStyleList*** | List of one or more segment styles. Only those styles included in the build’s SegmentStyleList may be referenced by the build |
| ***SegStyle*** | SegmentStyle ID to be applied to a particular segment. Name truncated to reduce XML file size |
| ***SkyWritingMode*** | Optional laser motion mode. If omitted or set to 0, will be disabled. See Scanlab RTC5 documentation for details |
| ***SpotSize*** | Laser spot size value in microns |
| ***Start*** | Starting coordinate of the first Segment of the Path   * All Segments are assumed to be contiguous from the End of the previous Segment, in contour fashion. Each Segment specifies only an End coordinate, rather than a separate Start/End for each Segment * To create non-contiguous hatches, set up alternating “mark” and “jump” segments by choosing a different SegmentStyles for each type of Segment |
| ***SyncDelay*** | Indicates the delay in microseconds between a particular Slave laser and the Master laser in “FollowMe” LaserMode. Should be omitted if LaserMode is set to Independent or if the SegmentStyle is a jump. Each Slave laser’s SyncDelay is absolute with respect to the Master, independent of any other Slave lasers which may be synchronized to the same Master. The Scanlab RTC5 supports delays in increments of 10 microseconds only |
| ***Trajectory*** | Grouping of related scan paths for one or more lasers   * Each TrajectoryID may contain multiple Paths, which may each be processed by the same or different lasers as controlled by SegmentStyles and PathProcessingMode * If there are multiple Trajectories within the layer, they will be processed sequentially. The first Trajectory must complete before the second can begin, irrespective of the lasers used by each Trajectory. Any elements which are to be built concurrently should be included in the same Trajectory |
| ***TrajectoryList*** | Contains all the scan paths for the layer. The individual trajectories will be built strictly sequentially in the order that they appear. Within a trajectory, however, individual paths (scan path groupings) may be built either sequentially or concurrently as defined by the trajectory’s PathProcessingMode value |
| ***Traveler*** | Section of a SegmentStyle which identifies and defines parameters for one specific laser. If a SegmentStyle utilizes multiple lasers, it should include multiple Traveler sections. Traveler:ID should be the system’s reference to a specific laser, such as “1” (serial numbers should not be used, to avoid machine-dependent files; a lookup from scanfile ID’s to serial numbers will be provided as part of the end-user machine controller) |
| ***Type*** | Indicates whether the Path consists of hatches or contours. Informational only; does not affect parameters |
| ***VelocityProfile*** | Metrics which defines a single mode of laser travel, including linear speed and various delays |
| ***VelocityProfile***  ***List*** | List of one or more velocity profiles. Only those profiles included in the build’s VelocityProfileList may be referenced by the build |
| ***Wobble*** | Optional mode of laser marking which adds an oscillating motion independent of laser travel speed. See Scanlab RTC5 documentation for details |