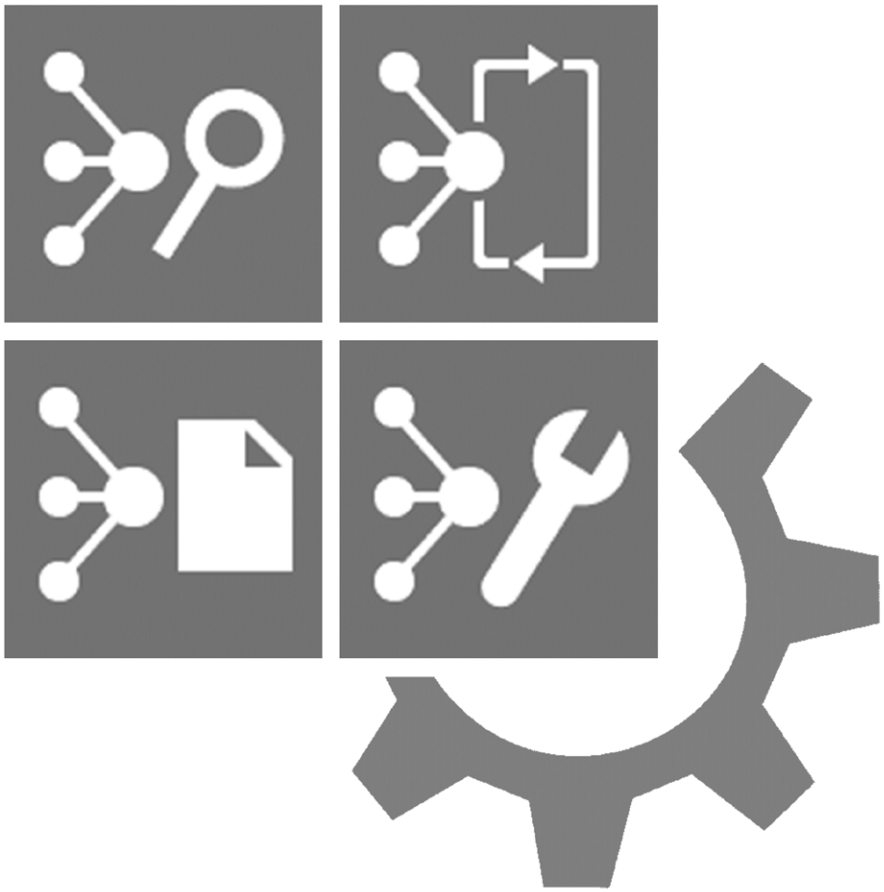
RENISHAW
apply innovation™

DataHUB developer manual



This page is intentionally left blank.

Contents

1 Before you begin

1.1 Terms and conditions and warranty

Unless you and Renishaw have agreed and signed a separate written agreement, the equipment and/or software are sold subject to the Renishaw Standard Terms and Conditions supplied with such equipment and/or software, or available on request from your local Renishaw office.

Renishaw warrants its equipment and software for a limited period (as set out in the Standard Terms and Conditions), provided that they are installed and used exactly as defined in associated Renishaw documentation. You should consult these Standard Terms and Conditions to find out the full details of your warranty.

Equipment and/or software purchased by you from a third-party supplier is subject to separate terms and conditions supplied with such equipment and/or software. You should contact your third-party supplier for details.

1.2 Changes to equipment

Renishaw reserves the right to change equipment specifications without notice.

1.3 Patents

Features of Renishaw's additive manufacturing systems, and other similar systems, are the subject of one or more of the following patents and/or patent applications

1.3.1 RenAM 500 series (Q, S and Flex models)

CA 2738618	EP 2331232	IN WO2014/125258	US 10335901
CA 2738619	EP 2875855	IN WO2014/125280	US 10493562
	EP 2956261	IN WO2014/199134	US 10500641
CN 102186554	EP 2956262		US 10639879
CN 105102160	EP 3007879	JP 6482476	US 10933620
CN 105228775	EP 3221073	JP 6571638	US 10974184
CN 105492188	EP 3221075		US 11033968
CN 107107193	EP 3299110		US 11040414
CN 107206494	EP 3323534		US 11104121
CN 107921659	EP 3325240		US 11267052
CN 108189390	EP 3357606		US 11305354
CN 108349005	EP 3377252		US 11478856
CN 108515182	EP 3377253		US 11565346
CN 109177153	EP 3566798		US 8753105
	EP 3689507		US 8794263
	EP 4023387		US 9114478
			US 9669583
			US 9849543
			US 2020-0023463
			US 2021-0354197
			US 2022-0203451
			US 2023-0122273

1.3.2 DataHUB

CN 109937101	EP 3482855	US 11167497	WO 2020/099852
CN 111315512	EP 3538295	US 2020-0276669	
CN 112996615	EP 3880391	US 2021-0394272	

1.3.3 InfiniAM Spectral

CN 105745060	EP 3049235	US 10850326	WO 2020/099852
CN 108349005	EP 3377252	US 11305354	WO 2020/174240
CN 109937101	EP 3482855	US 11040414	
CN 110026554	EP 3482909	US 2020-0276669	
CN 111315512	EP 3538295	US 2021-0039167	
CN 111491777	EP 3880391	US 2021-0394272	
CN 112996615	EP 3930999	US 2022-0168813	
CN 115943048	EP 2020-174240	US 2022-0203451	

2 Introduction

2.1 Scope of supply

This document is a guide to building, testing and deploying a plug-in software component for DataHUB. See the *DataHUB* user guide (Renishaw part no. H-5800-4761) for an overview and functional explanation of the DataHUB software. The plug-in development cycle begins with installing DataHUB on the development PC, then using Visual Studio to create a .NET assembly containing the plug-in code. With the appropriate Visual Studio debugging configuration, the plug-in can be tested and debugged to verify its correctness before its final deployment to a Data Collection PC (DCPC) in a production AM system.

2.2 Abbreviations

Term	Definition
AM	Additive Manufacturing
AMPM	Additive Manufacturing Process Monitoring
DCPC	Data Collection Personal Computer
HMI	Human Machine Interface (touch screen)
OEM	Original equipment manufacturer
PC	Personal Computer
PLC	Programmable Logic Controller
REACH	Registration, Evaluation, Authorisation and Restriction of Chemicals
WEEE	Waste Electrical and Electronic equipment

2.3 Safety information in this user guide

Within this user guide, additional information that is important to read and understand will be presented as a Warning, Caution or Note. The definitions of these are given below with examples.

2.3.1 Warning

An example of a Warning is as follows:

WARNING: A warning is to tell the end user that there is a possibility of injury to themselves or other people in the vicinity if the described course of action is not followed.

2.3.2 Caution

An example of a Caution is as follows:

CAUTION: A Caution is to tell the end user that there is a possibility of damage to the equipment if the described course of action is not followed.

2.3.3 Note

An example of a Note is as follows:

NOTE: A Note is to advise the end user of important information that is related to, or will assist them in the task or activity they are carrying out.

2.4 Training schedule

Renishaw provides a basic level of training to operate DataHUB safely. Renishaw also offers extended training courses for operators and process engineers. Refer to the *DataHub* user guide (Renishaw part no. H-5800-4761) and the training guide supplied as part of the user training course that all users should complete before using DataHUB.

2.5 Reference documentation

In addition to this User guide, also refer to the following documents for additional information about other aspects of the InfiniAM® Suite and Renishaw AM system.

- *RenAM 500Q/S additive manufacturing system* installation guide (Renishaw part no. H-5800-3692)
- *RenAM 500Q/S additive manufacturing system* user guide (Renishaw part no. H-5800-3693)
- *InfiniAM® and DataHUB software* installation guide (Renishaw part no. H-5800-4349)
- *DataHub* user guide (Renishaw part no. H-5800-4761)
- *InfiniAM® Spectral* user guide (Renishaw part no. H-5800-3919)
- *InfiniAM® Camera* user guide (Renishaw part no. H-5800-4675)
- Renishaw Licence Manager v2.x user manual (download via the link included in the InfiniAM licence email)
- Export Packets plug-in (Renishaw part no. 5800-6788)

3 Spare parts

There are no user-serviceable parts within DataHUB. In the event that DataHUB fails, then repair is by reinstallation and configuration of the software.

See Section 4, “Contact details”, for the contact details of your local Renishaw office and to arrange a service visit.

InfiniAM and DataHUB software will be periodically updated. All subscription users will be entitled to download the latest software release from their MyRenishaw.com account.

This page is intentionally left blank.

4 Contact details

Phone number	+44 (0) 1453 524524	
Hours of work	Monday to Thursday	08:00 to 17:00 (UTC and DST)
	Friday	08:00 to 16:00 (UTC and DST)
Email	am.support@renishaw.com	
Service address	Renishaw plc New Mills Wotton-under-Edge Gloucestershire GL12 8JR United Kingdom	
AM system type		
AM system serial number		
Software version numbers	HMI revision	
	PLC revision	
	PC revision	
InfiniAM Spectral hardware (MeltVIEW module) serial number		
InfiniAM software version number		
DataHUB software version number		

Please quote the details above. Details of your Renishaw AM system can be found on the serial plate on the rear of the AM system. Details of the MeltVIEW module can be found on the serial plate which is visible when InfiniAM is installed on the Renishaw AM system. Details of the CameraVIEW hardware can be found on the sticker situated on the rear of the camera. The CameraVIEW hardware can be found behind a cover above the chamber.

Additional support can be sought by contacting your local Renishaw office. See:

www.renishaw.com/contact

This page is intentionally left blank.

5 Safety

5.1 Introduction

WARNING: All safety information is in accordance with the applicable Renishaw AM system user guide and Renishaw AM system installation guide – unless otherwise stated within this document.

WARNING: Ensure that a blanking plate or the MeltVIEW hardware module is fitted to the laser aperture before turning on the AM system laser.

WARNING: The MeltVIEW hardware module is not interlocked to the laser safety circuit. If the MeltVIEW hardware module is removed and the laser is fired, harmful laser light will be emitted from the laser aperture on the AM system

5.2 DataHUB Generator specific laser and warning labels

There are no additional safety or warning labels fitted to the AM system as a result of installing DataHUB Generator.

This page is intentionally left blank.

6 Operation

6.1 Introduction

Through its plug-in architecture, the InfiniAM software suite supports the real-time analysis of process monitoring data collected by the LaserVIEW and MeltVIEW hardware platforms. A plug-in is a self-contained software unit installed on a Data Collection PC (DCPC) running DataHUB. When DataHUB processes the collected data of an additive manufacturing build, it passes process monitoring values to plug-ins for them to perform bespoke analysis. DataHUB supports multiple plug-ins running simultaneously, allowing diverse analysis algorithms and techniques to be applied to process monitoring data, and is integrated with the Renishaw Central system, allowing plug-ins to present analysis results alongside other machine sensor readings taken during the build.

DataHUB supports two plug-in API versions: version 1.0 (V1.0) supports plug-ins that process build data summarised as packets (see “Samples to packets” on page 6-8); version 2 (V2.0) supports plug-ins that process build data as token streams (see “Samples to token stream” on page 6-8). It is important to understand that the V2.0 API is not a replacement of the V1.0 API, rather they present the same process monitoring data to plug-ins, just in different formats. The choice of API to use should be carefully considered in relation to the analysis to be performed: the packets provided by V1.0 are considerably fewer in number than the samples provided by V2.0 and, as the fine detail of sample data may not be necessary for every analysis task, may be more suitable.

This document details the architecture of the plug-in system, the lifecycle of a plug-in, and other aspects of the system which apply to both API versions.

The Export Packets plug-in writes a tab-delimited file containing summary data of process monitoring data gathered by LaserVIEW and MeltVIEW hardware during a build. This plug-in is distributed with DataHUB and requires no extra licence.

6.2 Architecture

The high-level relationships between the hardware and software elements of the system are shown in Figure 1:

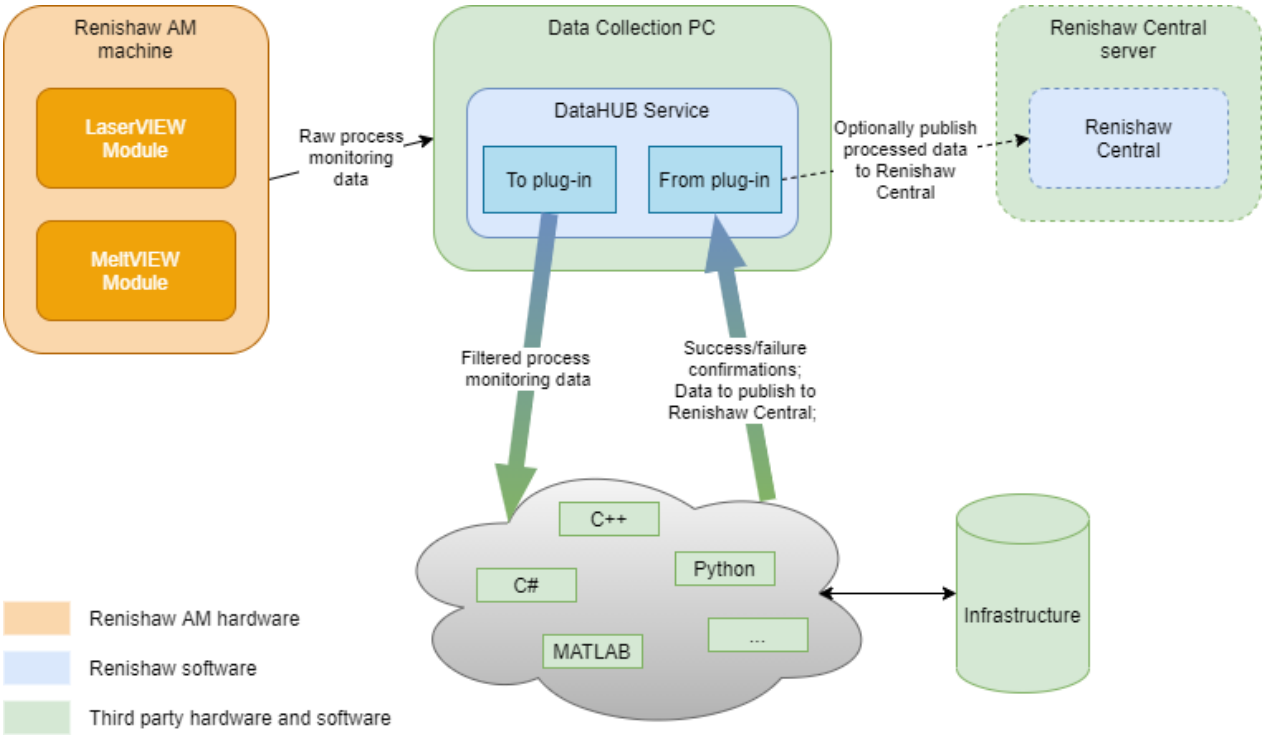


Figure 1 The DataHUB plug-in architecture

6.2.1 Data flow

As a Renishaw AM machine equipped with LaserVIEW and MeltVIEW hardware executes a build, process monitoring data is transferred to a DCPC. The DataHUB service running on that DCPC dispatches the data to all installed plug-ins. DataHUB runs an instance of each installed plug-in, per build. Each instance is allotted a uniquely named output folder for any persisted output, and each run independently, ensuring a failure in any one instance does not become a general failure of the system.

Figure 2 shows the typical data flow when a build is processed:

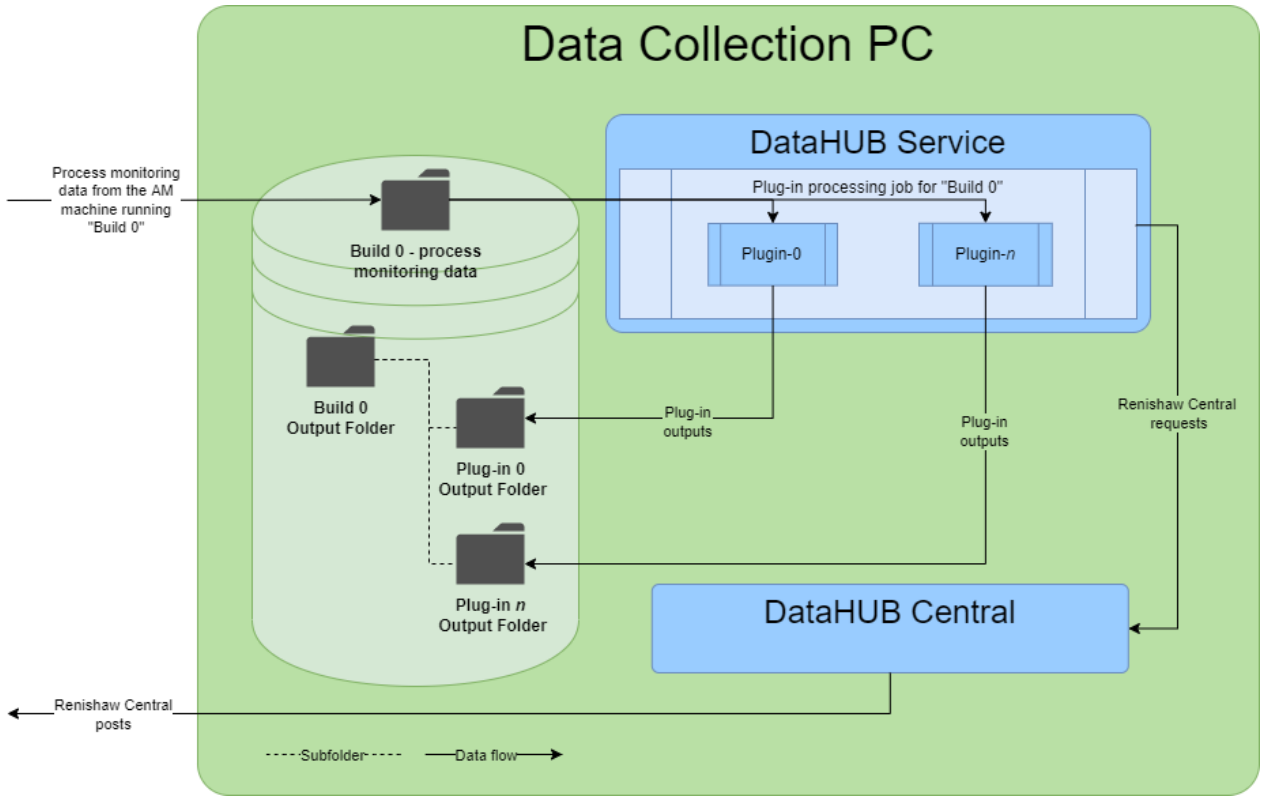


Figure 2 Build data flows

The process monitoring data gathered by an AM machine running a build is sent to a folder on the DCPC. The DataHUB service monitors this folder and, as new data arrives, pushes it out to the plug-in instances associated with the build job. Plug-ins may notify DataHUB service of results (time-series data, alerts, and analysis results) to be posted to Renishaw Central. The DataHUB Central service manages these upload requests.

6.2.2 Plug-in analysis technology

DataHUB is largely agnostic to the technology used to implement plug-ins and to any necessary supporting infrastructure. Its two requirements are as follows:

1. A plug-in is a .NET assembly implementing the set of methods defined as the plug-in API.

2. All runtime dependencies are installed on the DCPC.

DataHUB calls the plug-in API methods with values populated with process monitoring data, and the plug-in implementation is then at liberty to pass on the data to whatever technology best suits the analysis task in hand. A detailed explanation of integrating .NET with other technologies and programming languages is beyond the scope of this document, but a wealth of public domain information is available.

6.2.3 The “Push” data flow

The DataHUB service “pushes” data to plug-ins: the plug-in APIs are not for interrogating data sets and extracting data values. The typical use case where analysis is performed in real time as a build progress is best supported by this data flow. For archived data sets, DataHUB Generator may be used to instruct DataHUB service to reprocess the data so that additional plug-in analysis may be performed.

6.2.4 The plug-in lifecycle

For a plug-in to be used by DataHUB, it is first validated against each API’s implementation requisites, and only if found to conform will instances be created and used to analyse build data. Once validated, an instance of the plug-in is created for each new build data processing task. The lifecycle stages of a plug-in are shown in Figure 3:

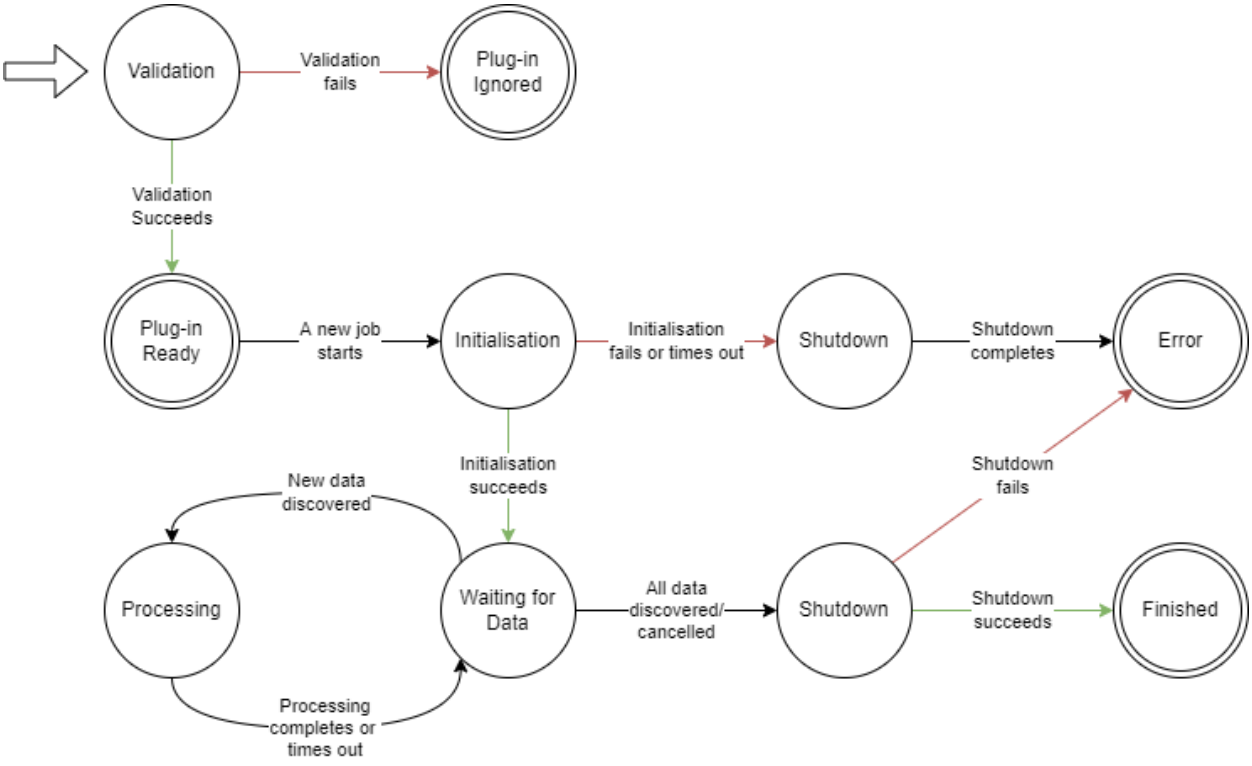


Figure 3 The various states a plug-in may pass through in its lifetime

6.2.4.1 Validation

When DataHUB starts, it searches for .NET assemblies that implement one or more public classes that conform to the DataHUB Plug-in APIs. All that conform are registered as valid plug-ins and will be instantiated whenever a “Process LaserVIEW and MeltVIEW via plug-ins” job is started.

An assembly may contain implementations of more than one plug-in, if so desired, including using different versions of the API.

6.2.4.2 Initialisation

When DataHUB starts a plug-in job, a new instance of a plug-in is created and its *Initialise* method called. The values passed to this method are “build constants”, for example, the number of lasers from which the plug-in may calculate resource allocation requirements, and so on. The method returns an indication that it successfully initialised or failed to initialise, and if it indicates failure, the instance *Shutdown* method is called immediately. The plug-in may optionally return definitions for time-series axes that will be populated by data points generated during later processing (see “Publishing data to Renishaw Central” on page 6-6).

6.2.4.3 Waiting for data/Processing

The data processing method is called as many times as is necessary to pass all the process monitoring data of a build to the plug-in instance. Each call, the plug-in will return an indication that it either successfully processed or failed to process the data. If it indicates a failure, the plug-in will not enter an error state – it will continue to have data passed to it (the assumption being that data for other layers remains of interest to the plug-in). The plug-in may optionally return time-series data points, alerts, and other analysis findings to be forwarded to Renishaw Central (see “Publishing data to Renishaw Central”).

6.2.4.4 Shutdown

The *Shutdown* method is called when no data remains to be processed (i.e., the build ran to completion, or was cancelled.) In this method the plug-in should perform any final processing on the build data, release any resources acquired, and so on.

6.2.5 Time

Time is communicated across the API in microseconds relative to the start of a layer. A time of 0 is coincident with the start of a layer, a time of 1000 is 1000 microseconds after the start of the layer and so on. Where necessary (for example, when posting to Renishaw Central) DataHUB will realise any relative times returned by a plug-in into an absolute time frame such as UTC.

6.2.6 Publishing data to Renishaw Central

If DataHUB has been connected to Renishaw Central, a plug-in can publish analysis results via DataHUB. If the returned string from the *Initialise* and Process methods conforms to the JSON schema recorded in the API definitions (see Section 6.4, “Plug-in API V2”), DataHUB will post the data contained to the Renishaw Central endpoint.

DataHUB recognises three types of uploads to Renishaw Central:

- Alerts
- Analysis results
- Time series

6.2.6.1 Alerts

An alert indicates that an important event has occurred at a point in time. An alert might be generated by a plug-in when it detects an anomaly in the build data, for example where the data implies a fault has occurred or will imminently occur in the build. Alerts come in three levels – *Info*, *Warning*, and *Error* – that are used to inform the viewing application (such as the Renishaw Central website) of the level of prominence to give the alert.

Alerts can only be raised during the *Process* lifecycle stage.

6.2.6.2 Analysis results

Analysis results, like alerts, may be returned in relation to analysis performed on build data. They have a flexible definition, allowing their content to be specialised dependent on the result type. This means that while (for example) the Renishaw Central website may not know how to represent every facet of an analysis result, a custom downstream application may be written that does.

Analysis results can only be raised during the *Process* lifecycle stage.

6.2.6.3 Time series

A time series represents a series of values across time. A time series can contain any kind of data, provided it can be plotted against time. During the *Process* stage, a plug-in may add two types of data to a time series: values and limits. Values are sets of time/value pairs that add a data point to the time series. Limits define expected ranges to aid display and indicate extremities in the data.

Time series are different from alerts and analysis results in that they must be first be defined before data points can be added. A plug-in may return time-series definitions from its *Initialise* method. Should the plug-in later attempt to add data to a time series that was not in this initial definition list (or if the definitions themselves were in some manner invalid), the data will be rejected and lost.

6.2.7 Audit logging

DataHUB maintains a set of logs that record performed actions, including some that may be used to diagnose issues with a plug-in. The logs are located at the location:

<$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>

Logs are named for the day they were recorded using the format “yyyyMMdd.”

The key entries that will interest a plug-in developer are as follows:

- On start-up, DataHUB records:
  - version numbers
  - licencing details
  - the validation of plug-in assemblies
- On the start of a build, DataHUB records:
  - plug-in instance creation
  - the values used to initialise each plug-in instance
  - the *Initialise* method result returned
- During a build, DataHUB records:
  - non-trivial results from plug-in *Processing* calls (in the interest of reducing logging spam, a plug-in may return a trivial result which is not logged)
- When a build completes, DataHUB records:
  - the result of each plug-in *Shutdown* method call

6.2.8 Process monitoring data pre-processing

The LaserVIEW and MeltVIEW modules gather data as a stream of samples, each associated with a specific laser, a position on the powder bed, and various sensor readings, at a particular time. DataHUB performs some pre-processing of this “raw” process monitoring data. For V1.0 plug-ins, DataHUB summaries samples into packets, and for V2.0 the sample data is assembled into a token stream.

6.2.8.1 Samples to packets

The DataHUB service quantises consecutive samples that share the same position, with the laser firing, into a packet. A packet’s start time is taken from its first contributing sample, and the final contributing sample time is used to calculate the duration of the packet. The LaserVIEW and MeltVIEW sensor reading values of the contributing samples are averaged.

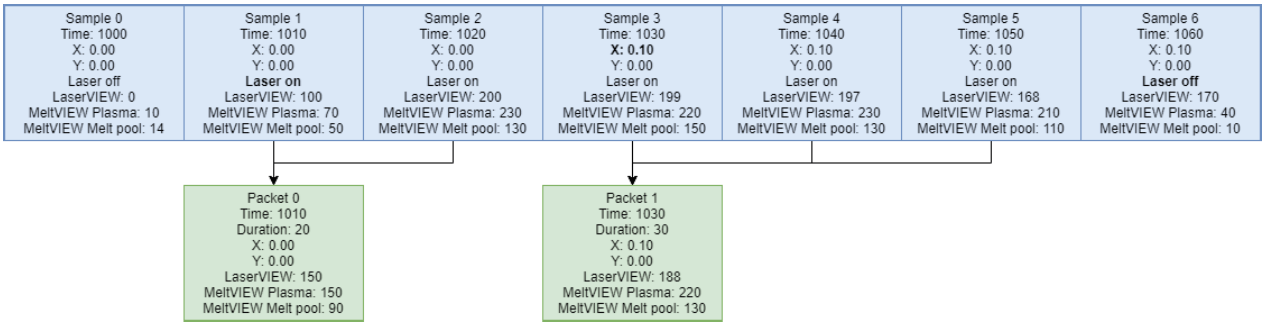


Figure 4 Samples to packets

In Figure 4, seven samples are quantised into two packets. The first sample is ignored, as the laser is not firing. As they share the same position and the laser is firing, samples 1 and 2 are quantised into a packet. The position of sample 3 changes from its predecessor, therefore a new packet begins, gathering samples until the laser stops firing. The final sample 6 is not added as the laser is off, though it shares the same position as sample 5.

6.2.8.2 Samples to token stream

When converting the process monitoring samples into a token stream, DataHUB builds a set of *Sample* tokens surrounded by other information tokens, the whole describing the structure and content of the build. Unlike V1.0, samples taken while the laser was not firing are considered and so added to the token stream.

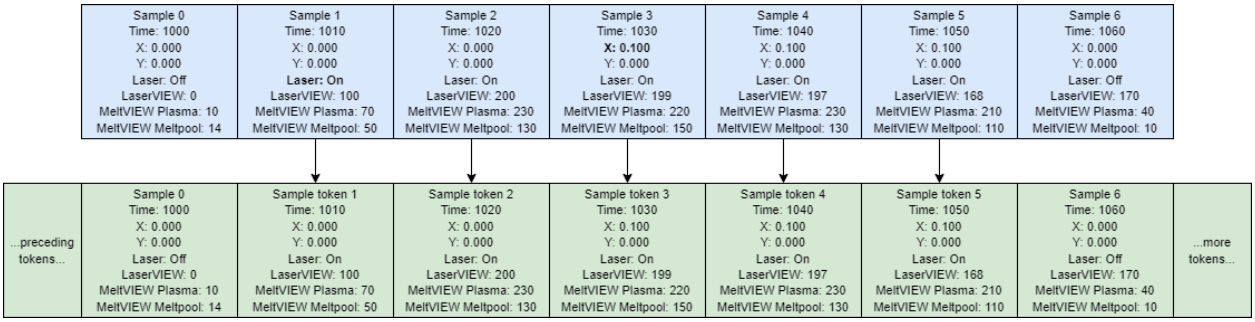


Figure 5 Samples to token stream

Figure 5 shows a portion of a token stream given the same samples as used in *Samples to packets*, above.

6.3 Plug-in API V1

This document is a guide to building, testing and deploying a V1.0 plug-in software component for DataHUB. The plug-in development cycle begins with installing DataHUB on the development PC, then using Visual Studio to create a .NET assembly containing the plug-in code. With the appropriate Visual Studio debugging configuration, the plug-in can be tested and debugged to verify its correctness before its final deployment to a Data Collection PC (DCPC) in a production AM system.

6.3.1 Prerequisite activities

The following documents should be read in conjunction with this document:

- *InfiniAM® and DataHUB software* installation guide (Renishaw part no. H-5800-4349)
- *DataHUB* user guide (Renishaw part no. H-5800-4761)
- Renishaw Licence Manager v2.x user manual

Before proceeding, the following activities should be performed on the PC to be used for plug-in development:

- Ensure the PC is equipped with a GPU at least at the DataHUB minimum specification (see the data collection PC hardware specification in the *InfiniAM® and DataHUB software* installation guide).
- Verify up-to-date drivers for the GPU are installed.
- Ensure the licence server has sufficient appropriate licences for DataHUB and Spectral API.
- Verify network access to the licence server.
- Ensure a version of Microsoft Visual Studio is installed that supports .NET Framework 4.7.2.

6.3.2 Installing DataHUB for plug-in development

The DataHUB service must be installed on the development PC. During installation, the option of which licences to claim is offered. Ensure the “LaserVIEW/MeltVIEW Pluggability” option is checked:

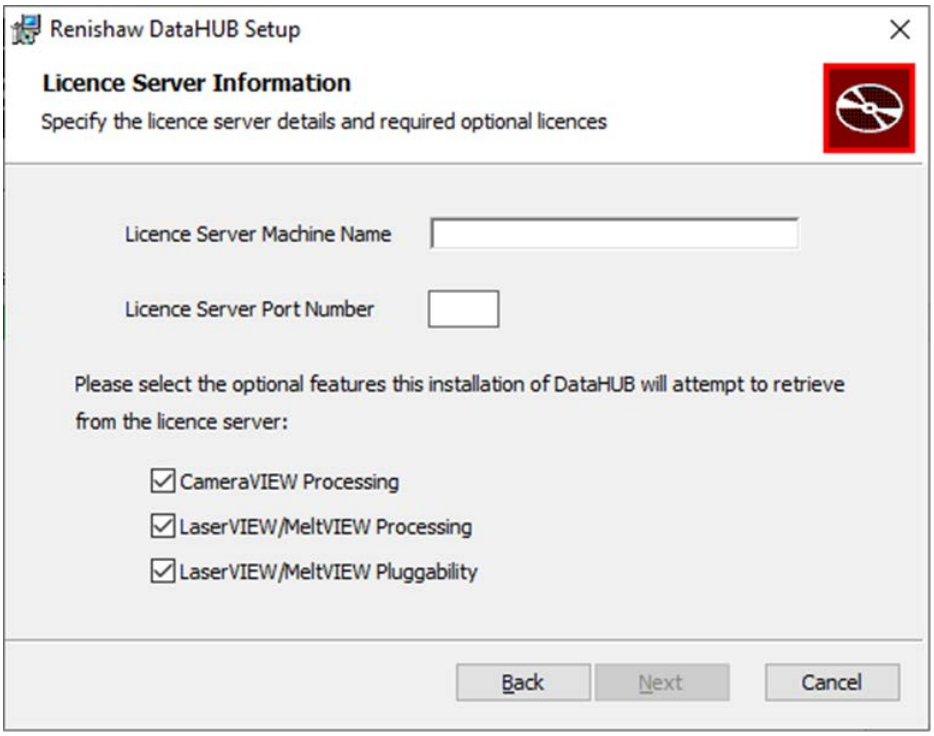


Figure 6 Licensing DataHUB

6.3.3 Implementing the plug-in

The following sections of this document demonstrate the end-to-end workflow of creating, compiling, debugging and deploying a plug-in using Visual Studio.

Plug-ins are written in C# and define a public class implementing the specific set of public methods DataHUB uses to identify, validate and use instances of the plug-in when a build's process monitoring data is processed.

NOTE: The screenshots shown below were taken from Microsoft Visual Studio 2019, but the workflow is broadly similar in other versions.

6.3.3.1 Creating the Visual Studio solution

Start Visual Studio and choose *Create a new project*. From the available options, choose *Class Library (.NET Framework)*:

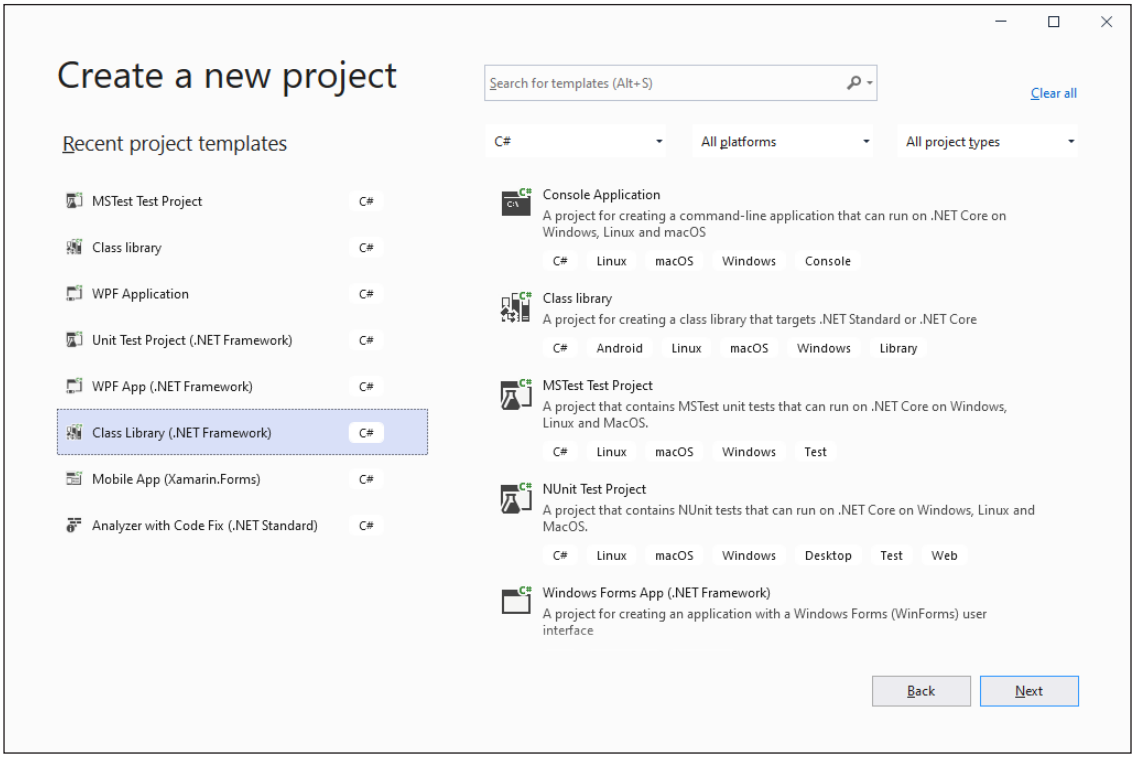


Figure 7 Creating the solution

Next, configure the project:

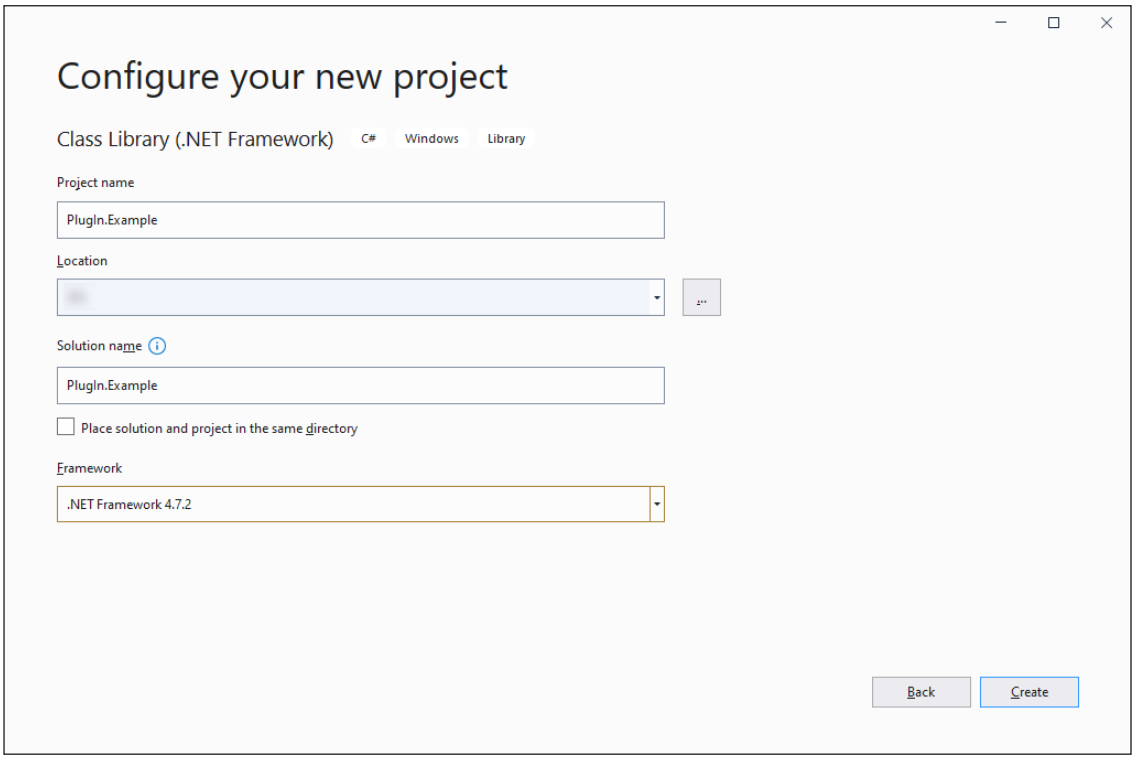


Figure 8 Configuring the project

DataHUB requires a plug-in assembly must start with **PlugIn.** (the word “PlugIn” followed by a full-stop). Select a folder of your choice as the project’s *Location* and create the solution.

NOTE: DataHUB supports plug-ins targeting x86/x64/Any CPU. It is recommended to target Framework 4.7.2 or higher.

6.3.3.2 Naming the plug-in type

The single default source file Class1.cs contains the definition of `Class1`, and we will use this as the starting point.

```
namespace PlugIn.Example
{
    public class Class1
    {
    }
}
```

Once DataHUB discovers a potential plug-in assembly, it searches for a public type `PlugIn`, so rename `Class1` as `PlugIn`.

```
namespace PlugIn.Example
{
    public class PlugIn
    {
    }
}
```

6.3.3.3 Adding the API

With a potential plug-in type, DataHUB requires it implements the following public methods:

`public PlugIn()` (generated automatically)
`public string APIVersion()`
`public string DisplayName()`
`public string Initialise(…)`
`public string ProcessPackets(…)`
`public string Shutdown()`

Add the following method implementations to the class:

```
public class PlugIn
{
   public string APIVersion() => “”;
   public string DisplayName() => “”;
   public string Initialise(
      string plugInFolderPath,
      string outputFolderPath,
      uint numberOfDatFilesPerLayer) => “”;
```

```
    public string ProcessPackets(
        uint layerNumber,
        uint laserId,
        int count,
        uint[] startTime,
        uint[] duration,
        double[] demandPositionX,
        double[] demandPositionY,
        uint[] laserVIEW,
        uint[] meltVIEWPlasma,
        uint[] meltVIEWMeltpool) => "";
    public string Shutdown() => "";
}
```

Notice all API methods return a `string` (which for the moment have been left empty.) The content of the returned string is used to signal to DataHUB the result of the method. As each method is fleshed out, below, the role of this return value will be explained.

6.3.3.4 The APIVersion method

This method reports to DataHUB the version of the API used by the plug-in. To use the API version 1.0, this must return the literal "1.0":

```
public string APIVersion() => "1.0";
```

6.3.3.5 The DisplayName method

The content of the string returned by this method is used as the plug-in name displayed in DataHUB Monitor, and in creating the plug-in's output folder. Although it is not mandatory for this name to be unique, making it so helps to identify the plug-in when, for example, multiple versions are installed on a DCPC.

```
public string DisplayName() => "Example plug-in";
```

6.3.3.6 The Initialise method

This method is called by DataHUB at the start of a build, before any layer data is sent to the plug-in. Hence it passes *build invariant* data to the plug-in. In the example plug-in, the *Initialise* method assigns the path to the instance output folder to the field **_outputFolderPath**:

```
public string Initialise(
   string plugInFolderPath,
   string outputFolderPath,
   uint numberOfDatFilesPerLayer)
{
   _outputFolderPath = outputFolderPath;
   return “Success”;
}
…
private string _outputFolderPath;
```

6.3.3.7 The ProcessPackets method

This method is called by DataHUB when all data for a layer has been received. In the example plug-in, the *ProcessPackets* method first builds a line of delimiter-separated columns headings, then adds line-by-line the values for the process monitoring packet data:

```
public string ProcessPackets(
   uint layerNumber,
   uint laserId,
   int count,
   uint[] startTime,
   uint[] duration,
   double[] demandPositionX,
   double[] demandPositionY,
   uint[] laserVIEW,
   uint[] meltVIEWPlasma,
   uint[] meltVIEWMeltpool)
{
   var builder = new StringBuilder().
      Append($”Start time / us\t”).
      Append($”Duration / us\t”).
      Append($”x position / mm\t”).
      Append($”y position / mm\t”).
      Append($”LaserVIEW\t”).
      Append($”MeltVIEW Plasma\t”).
      Append($”MeltVIEW Melt Pool”).
```

```
        Append(Environment.NewLine);
    for (var i = 0; i < count; ++i)
        builder.
            Append($"{startTime[i]}").
            Append($"\t").
            Append($"{duration[i]}").
            Append($" \t").
            Append($"{demandPositionX[i].ToString("F3", CultureInfo.CurrentCulture)}\t").
            Append($"{demandPositionY[i].ToString("F3", CultureInfo.CurrentCulture)}\t").
            Append($"{laserVIEW[i]}\t").
            Append($"{meltVIEWPlasma[i]}\t").
            Append($"{meltVIEWMeltpool[i]}").
            Append(Environment.NewLine);
    System.IO.File.WriteAllText(
        Path.Combine(_outputFolderPath,
                    $"Packet data for layer {layerNumber}, laser {laserId}.txt"),
        builder.ToString());
    return "Success";
}
```

6.3.3.8 The Shutdown method

This method is called by DataHUB when all data for a build has been sent to the plug-in, or when any previous call to the plug-in failed. The method has no parameters.

In this method, the plug-in should perform any final processing on the build data, release any resources it may have acquired, and so on. In the example plug-in, the *Shutdown* method is:

```
public string Shutdown() => "Success";
```

6.3.4 Deploying the plug-in for debugging

Build the solution in *Debug* and open a file browser and locate the output folder which will contain a *.dll* file named as per the name given to the project, for example, *PlugIn.Example.dll*. (Ancillary files such as debug symbols, etc. may also be present.)

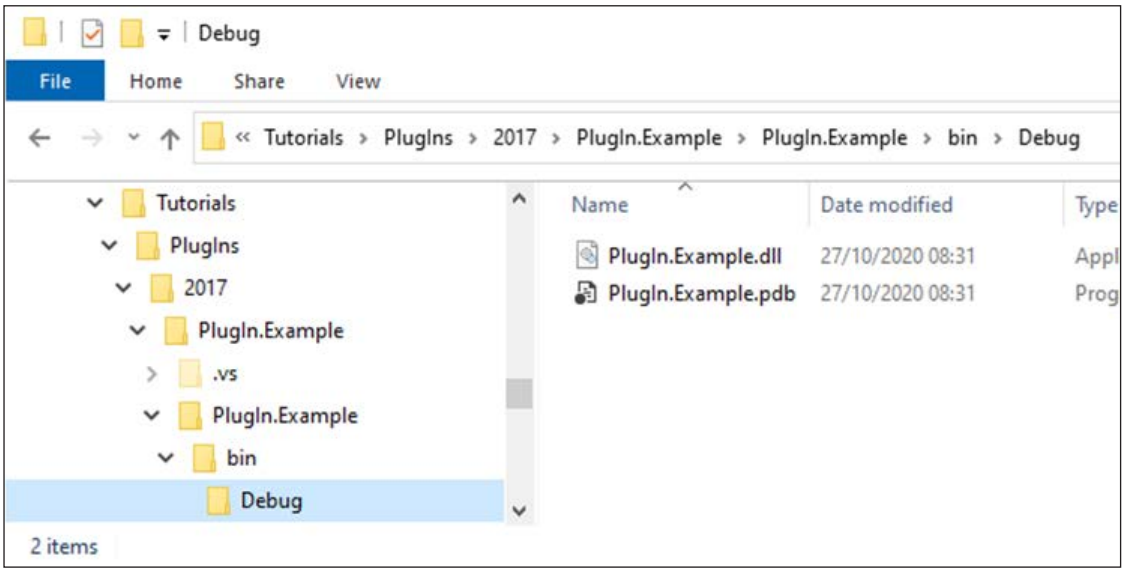


Figure 9 The plug-in binaries

Copy this folder and paste it into the *PlugIns* subfolder in the data folder of DataHUB:

<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>.

NOTE: Typically, $PROGRAMDATA$ will be located at <C:\ProgramData>, but may be hidden, in which case the Windows File Explorer application option “View, Show/hide, Hidden items” option should be checked.

Administrator privileges may be required to modify the contents of this folder.

Rename the *Debug* folder to suit the name of the plug-in – in the image below, the folder has been named *PlugIn.Example*.

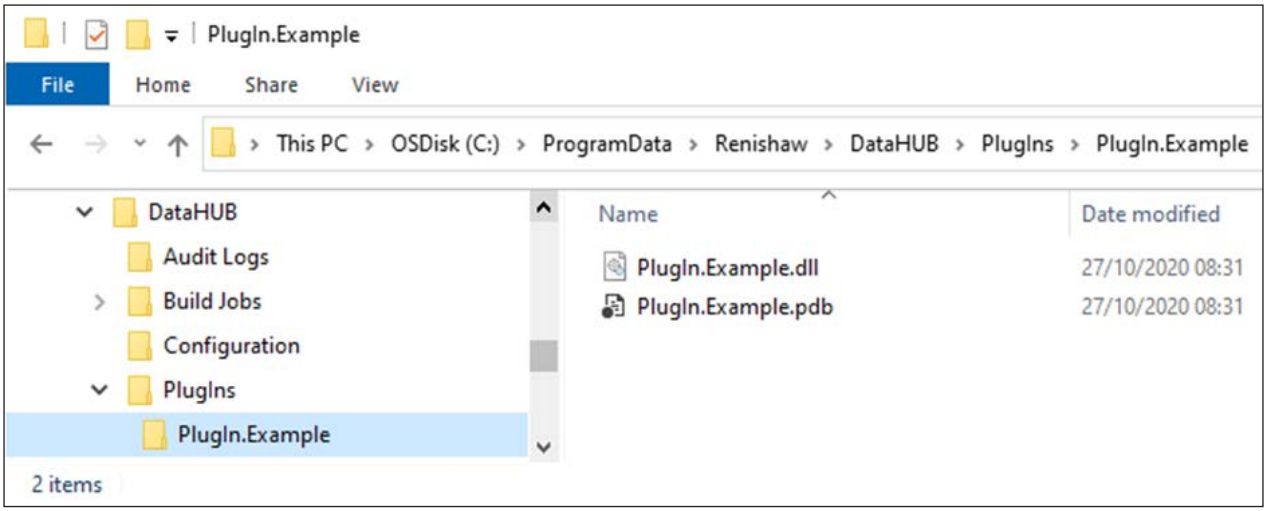


Figure 10 The plug-in deployed

6.3.4.1 Registering the plug-in

DataHUB service must be restarted to register the new plug-in. Use the *Services* application to do this, by right-clicking on *DataHUB Service* and choosing *Restart*:

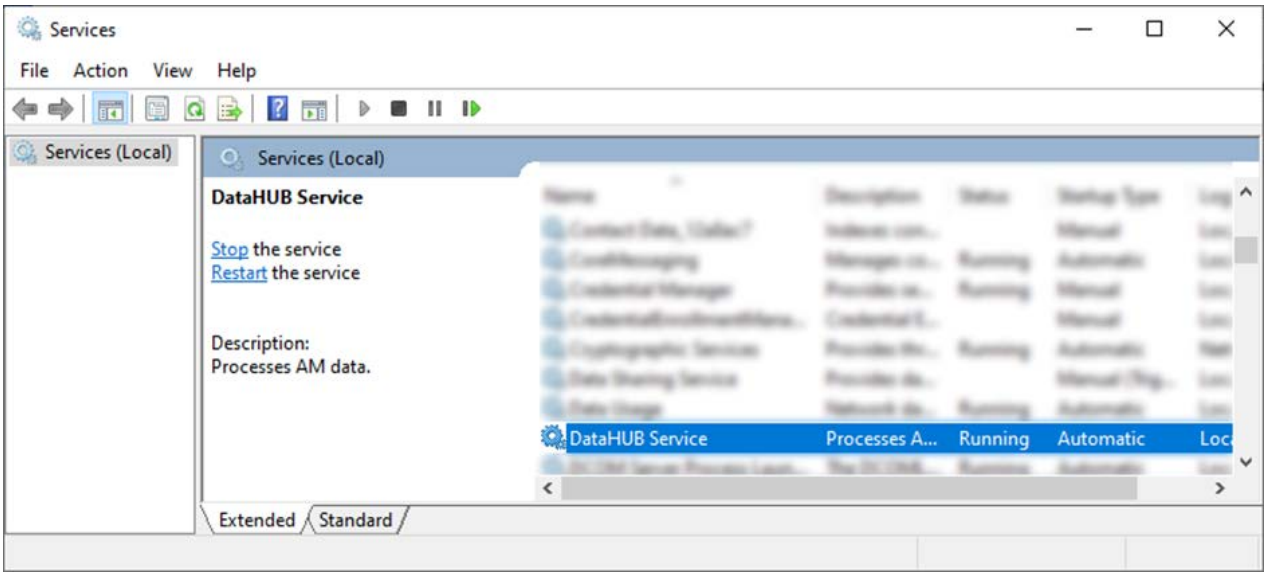


Figure 11 The Windows Service application

After a few seconds, the DataHUB service will show its status as *Running*.

6.3.4.2 Checking registration

DataHUB generates a log file as it runs, detailing important events such as build starts, build errors and, of course, plug-in auditing. Using File Explorer, locate the log folder <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>.

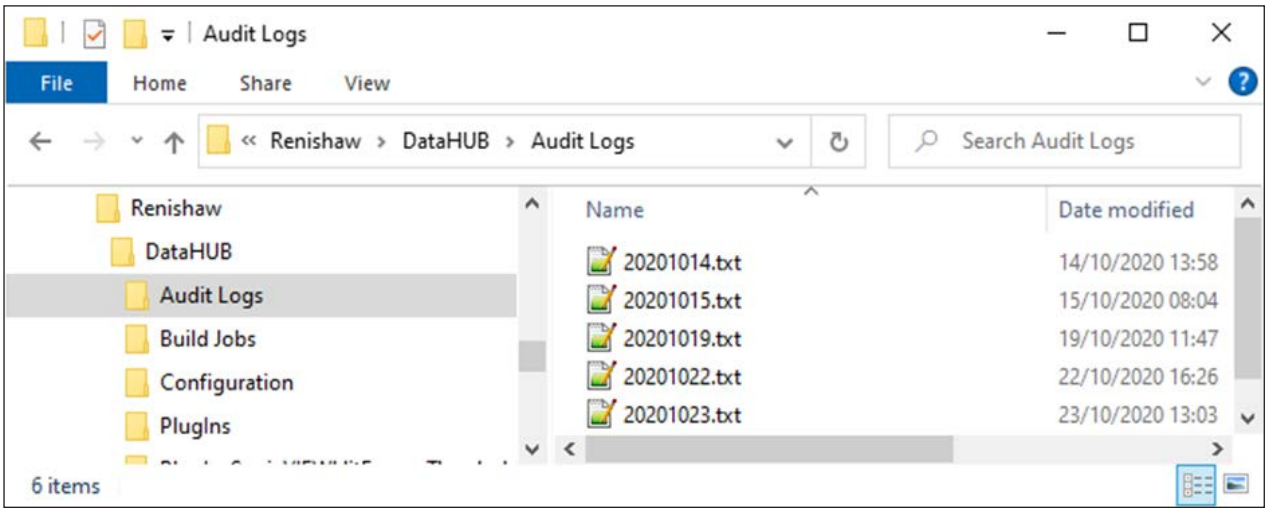


Figure 12 The DataHUB log folder

Open the most recent log file – it should have today's date – scroll to the end and look for lines detailing the discovery of valid plug-ins:

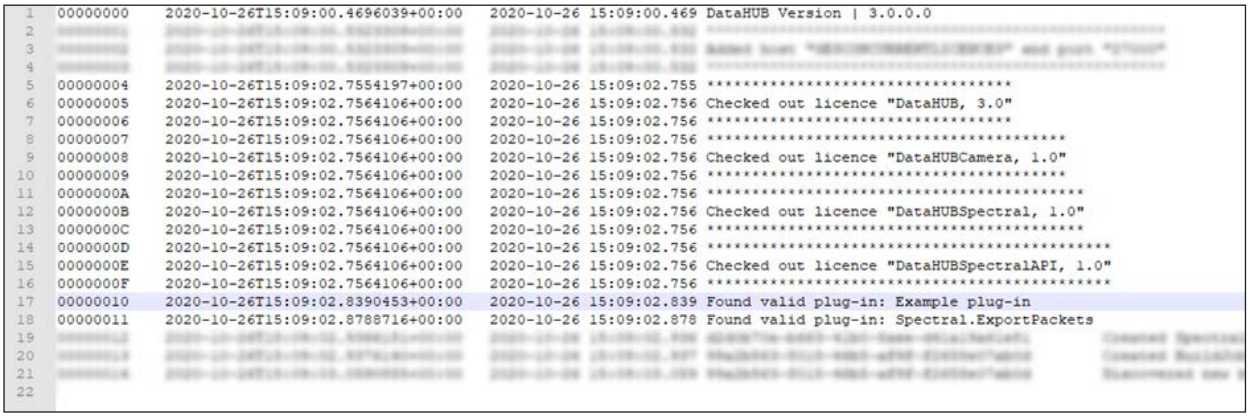


Figure 13 Valid plug-ins found by DataHUB

6.3.5 Timeout

As DataHUB cannot guarantee every call to a plug-in instance will return in a timely fashion, it adopts a timeout policy. If a plug-in call fails to complete within a certain duration, it is presumed faulty and shut down. The default timeouts are:

- For *Initialise*, 20 seconds
- For *ProcessPackets*, 200 seconds
- For *Shutdown*, 20 seconds

During debugging it will often be the case that these timeouts will be exceeded, causing DataHUB to shut down the plug-in prematurely. To allow for more debugging time, the timeout durations can be increased by editing the file <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> in a suitable XML or text editor.

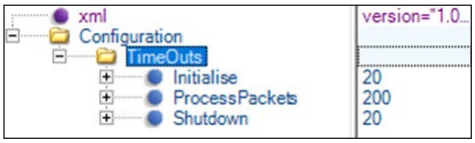


Figure 14 The default plug-in timeouts

NOTE: The DataHUB service must be restarted for these timeout duration changes to take effect.

6.3.6 Debugging the plug-in

With a successfully deployed valid plug-in, it is now possible to debug its execution. The plug-in itself is not directly executable, but by attaching the Visual Studio debugger to the DataHUB service and starting a build job, the plug-in implementation will be invoked by the service, and breakpoints set in the code will be hit.

NOTE: To debug via the service, Visual Studio may require administrator privileges. These can be granted by right-clicking the Visual Studio program shortcut and selecting “Run as administrator”.

In Visual Studio, set breakpoints at the start of the *Initialise* and *ProcessPackets* methods:

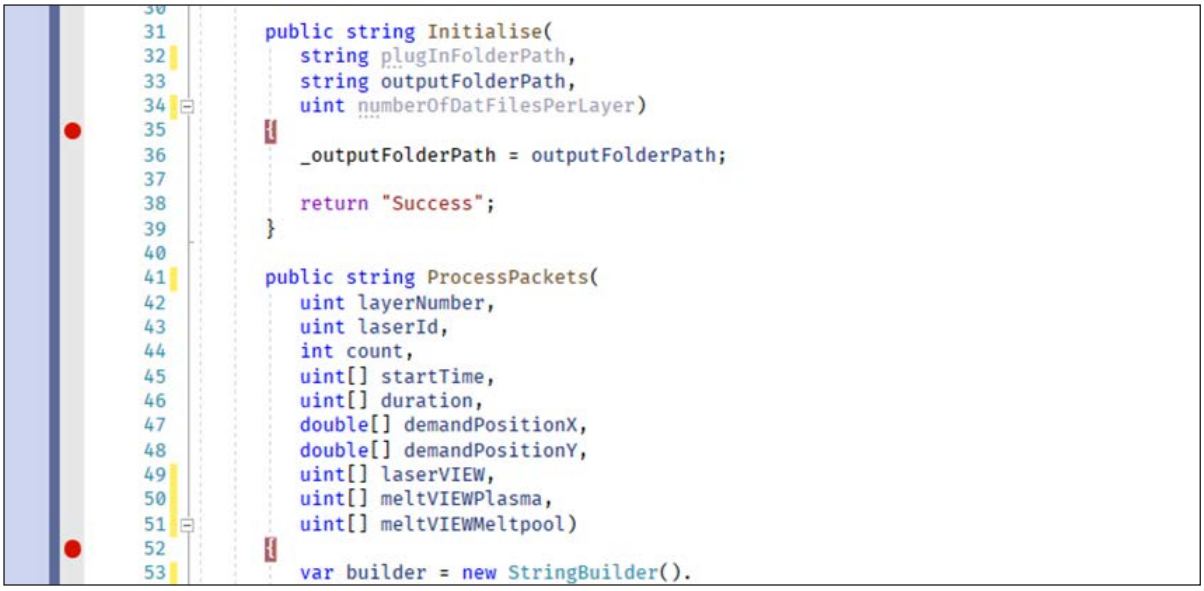


Figure 15 Breakpoints set in *Initialise* and *ProcessPackets*

Open the Debug menu and select “Attach to process…”:

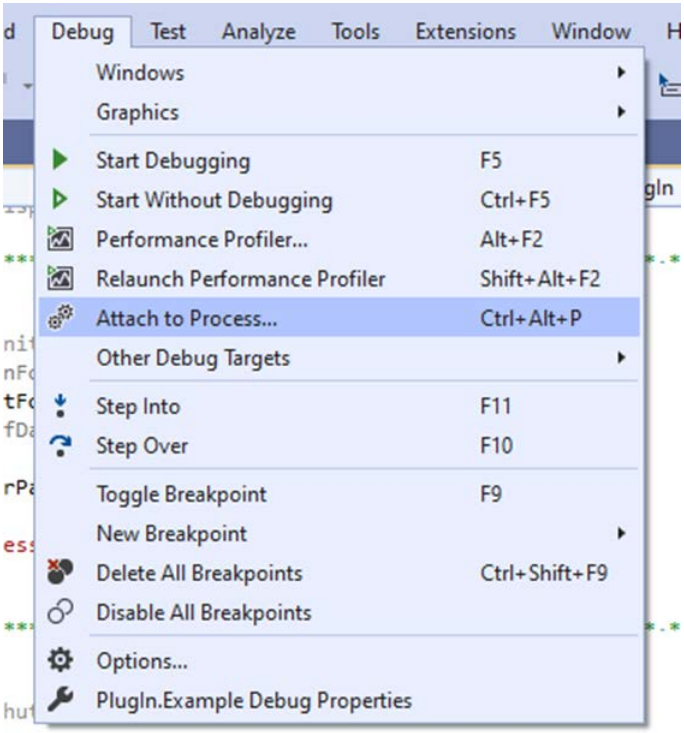


Figure 16 Attaching to a process for debugging

In the list of running processes, locate *DataHUB Service.exe* – you may need to check “Show processes from all users”:

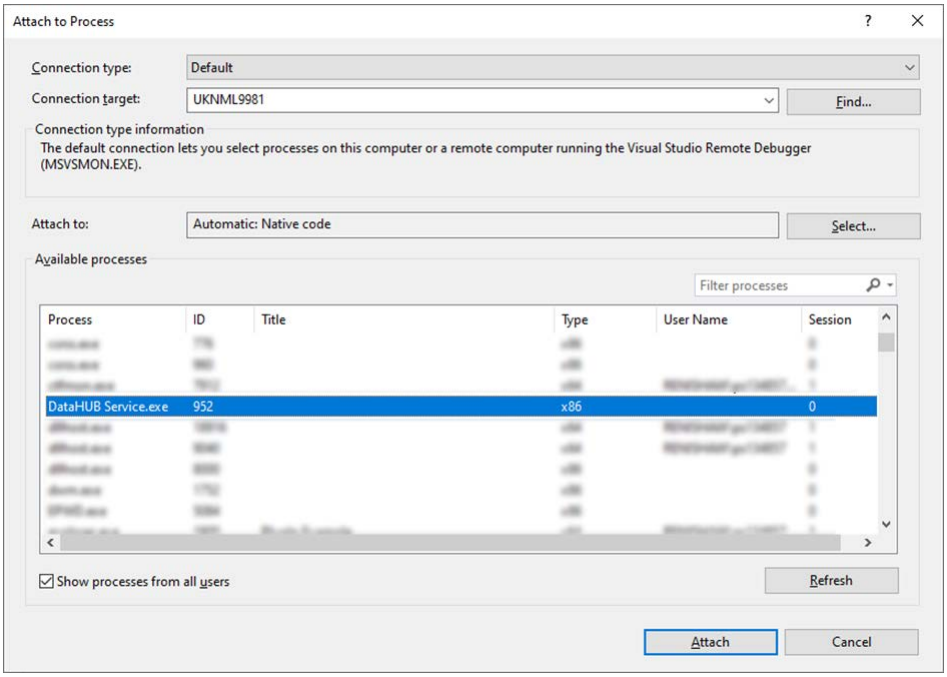


Figure 17 The DataHUB Service process

Press the “Attach” button, and Visual Studio will begin a debugging session against the DataHUB service.

For DataHUB to call the plug-in and hence for the breakpoints to be hit, a build must be started. This can be done via DataHUB Generator, or, if you have already generated some builds, simply by duplicating a build folder in <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs>.

DataHUB will automatically detect the new build and begin processing data. One of the service's preliminary tasks is to create (by invoking the parameterless constructor) an instance of each registered plug-in. When DataHUB then calls the instance *Initialise* method, the first breakpoint will be hit. The values of the method parameters will be those of the build in question, for example:

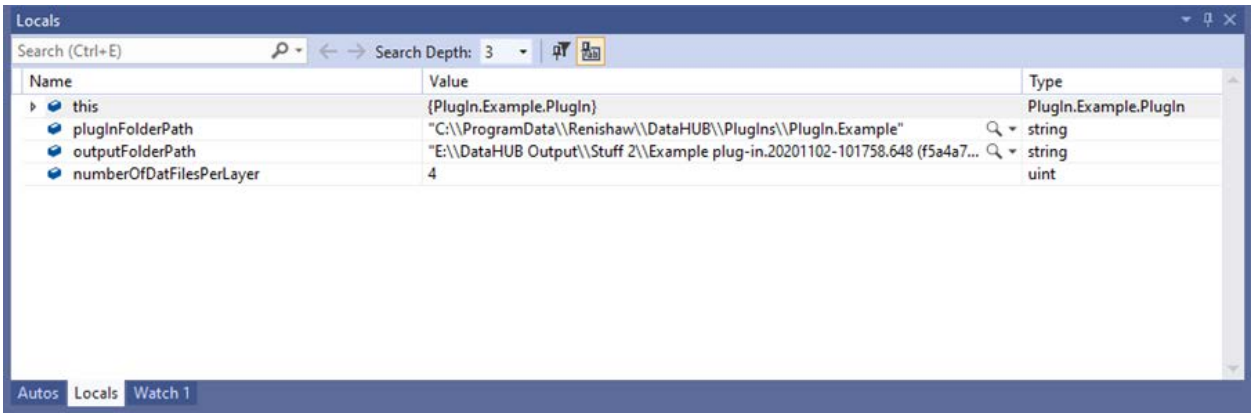


Figure 18 *Initialise* method parameter values

From then on, existing data (and, for a live build, new data as it arrives, is processed and the *ProcessPackets* method of the instance is called with parameter values for the LaserVIEW and MeltVIEW data associated for a layer:

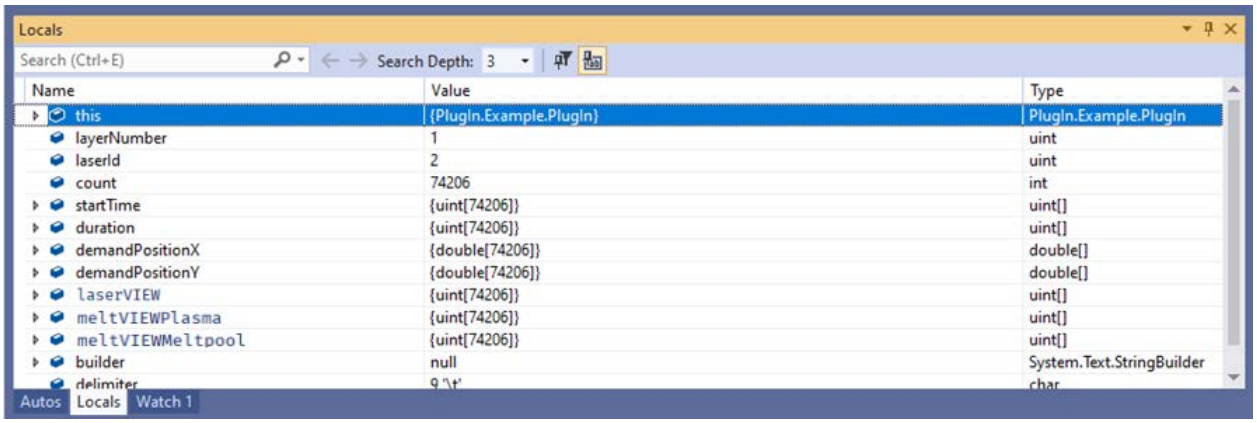


Figure 19 *ProcessPacket* method parameter values

The exercise of adding and checking breakpoints in the *Shutdown* method is left to the reader.

6.3.7 Deploying the plug-in for production

In preparation for deploying the plug-in to a production DCPC, it is critical that a Release build be compiled, locally deployed and tested. Once the functionality of the plug-in is validated and verified, it may be deployed to production DCPCs.

The process for deployment to a production DCPC is very similar to that of deploying to a development PC. The *Release* folder should be copied to the *PlugIns* subfolder in the data folder of DataHUB (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) on the DCPC and renamed to suit the name of the plug-in. The DataHUB service then must be restarted to register the new plug-in: use the *Services* application to do this. The success or otherwise of registering the new plug-in is logged in the day's audit log.

NOTE: DataHUB is designed to resume all in-progress data processing tasks after a restart, so it is not necessary to wait for all in-progress tasks to complete before restarting. However, only new data processing tasks will use the newly registered plug-in.

6.3.8 Diagnosing plug-in failures in production

In the production environment, DataHUB Monitor will show (alongside other build processing tasks) the processing status for the plug-in:

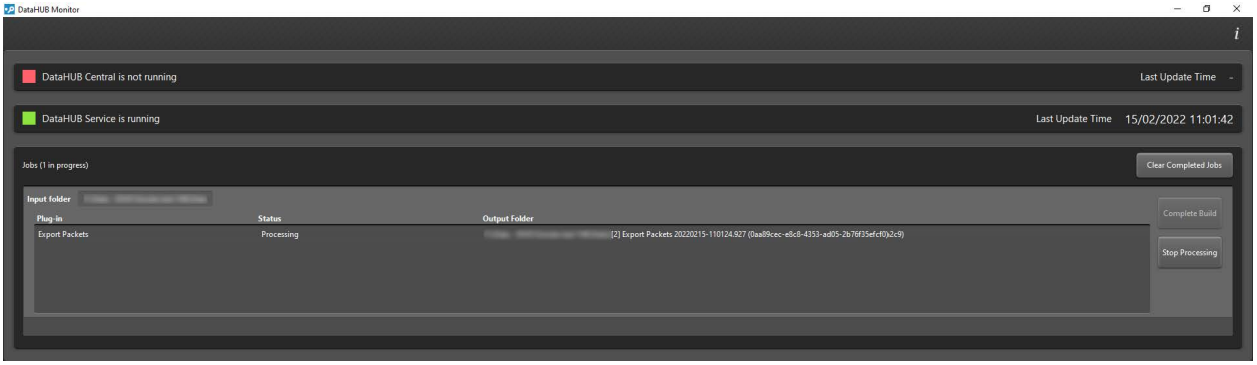


Figure 20 A build running with plug-ins

If the status of a plug-in is *Error*, the audit log will contain records of the failure, either as entries added by DataHUB itself (for example, the failure to load the plug-in assembly due to missing dependencies), or by the plug-in via the contents of the return value from the plug-in's *Initialise*, *ProcessPackets* or *Shutdown* methods.

6.3.9 Troubleshooting

The following table shows common failures, their typical audit log entry, and potential fixes:

Audit log entry	Failure	Potential fix
The assembly does not contain a class named PlugIn	PlugIn class not found	Rename the plug-in class, recompile, and redeploy
The PlugIn class does not contain a method named xyz	PlugIn class does not implement one of the API methods	Examine the class for missing or misspelt methods
The xyz method does not have the expected signature …	PlugIn class does not correctly implement one of the API methods	Examine the method signature and ensure all parameters match the API definition
The PlugIn class is using an unsupported version of the plugin API '{version}'. This version of DataHUB only supports version '1.0' of the plugin API.	PlugIn class does not return a valid API version string	Ensure the APIVersion method returns "1.0"
Failed to create plug-in output folder <path>	The folder for a specific instance of a plug-in was not created	Ensure read/write/modify/create access rights to <path>
Initialisation result: timed out Process result: timed out Shutdown result: timed out	A call to one of these API methods took too long to return: the plug-in is presumed failed	Consider reducing the amount of work done in the method, or increase the timeout duration in PlugInsConfiguration.xml
Exception:	An unexpected error occurred	Examine the logged exception messages to determine the source of the failure

6.3.10 Renishaw Central integration

Analysis results (time-series data points, alerts, and so on) of a plug-in can be forwarded to a Renishaw Central server. *Plug-in API V2.0 and tutorials* (Renishaw part no. H-5800-6784) contains a full explanation of how to structure the content of the string returned by the plug-in's *Initialise* and *ProcessPackets* methods to achieve this integration, as well as details of helper functions in the *PlugIns.API.v2.dll* that are compatible with the V1.0 API.

6.3.11 The Plug-in API V1.0

6.3.11.1 Assembly naming

The .NET assembly file name must start with “PlugIn.” (“PlugIn” followed by a full-stop) and have the extension “dll”.

6.3.11.2 Plug-in class naming

The plug-in assembly must define a public class named “`PlugIn`”.

6.3.11.3 Plug-in class methods

The plug-in class must implement the following public methods:

```
public PlugIn()
public string APIVersion()
public string DisplayName()
public string Initialise(…)
public string ProcessPackets(…)
public string Shutdown()
```

6.3.11.4 The parameterless constructor

The type must have a parameterless constructor. If no constructors with parameters are defined, C# provides this automatically, i.e., there is no need to define it explicitly.

NOTE: A plug-in should not acquire any resources inside its constructor.

A plug-in instance that is constructed is not guaranteed to have its *Shutdown* called, therefore any such resources may not be cleaned up in a timely manner. *Initialise* is the appropriate place to acquire resources.

6.3.11.5 The `APIVersion` method

This identifies which version of the API this plug-in should be validated against, and the format of the process monitoring data it will process.

Parameters:

None.

Return: `string`

A plug-in that implements API V1.0 must return the literal “1.0”.

6.3.11.6 The `DisplayName` method

The content of the string returned by this method is used as the plug-in name displayed in DataHUB Monitor, in creating the plug-in's output folder, and when auditing events for the plug-in. Although it is not mandatory for this name to be unique, making it so helps to identify the plug-in when, for example, multiple versions are installed on a DCPC.

Parameters:

None.

Return: `string`

The literal to use in various parts of the DataHUB UX, logging, and so on.

6.3.11.7 The `Initialise` method

This method is called by DataHUB at the start of a build, before any layer data is sent to the plug-in, hence it receives *build invariant* parameter values. The plug-in may use this method to allocate resources required during the lifetime of the instance, calculate build constants, and so on.

Parameter 1: `string`

The path of the folder of the plug-in assembly, e.g., <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugIn.Example>

Parameter 2: `string`

The path of a folder into which this plug-in may write output files. Note that for each plug-in, for each build, this folder will be unique and named using the plug-in's display name (as defined by the value returned by the `DisplayName` method), a timestamp, and a GUID. Each time the plug-in is run for a build, the folder name will change, but it always uses this structure:

<Build output folder>\[1.0] <Plug-in display name>.yyyyMMdd-HHmmss.fff (GUID)

Parameter 3: `uint`

This will be a value from 1 to 4, as defined by the hardware configuration of the machine creating the data.

Return: `string`

The plug-in should return a string containing either:

- the word "Success", if the method completed correctly, or
- detail as desired on the cause of failure, or
- a JSON string conforming to the Renishaw Central return API.

If DataHUB receives a non-conforming string, the plug-in will be shut down, and the content of the returned string added to the audit log for later diagnosis.

6.3.11.8 The `ProcessPackets` method

This method is called by DataHUB when all data for a layer has been received on the DCPC. The parameters of this method are:

Parameter 1: `uint`

The number of the build layer to which the data relates. The first layer is layer 1.

Parameter 2: `uint`

The identifying number, “1”, “2”, “3” or “4”, of the laser for which the data was gathered.

Parameter 3: `int`

The count of the number of elements in the arrays following.

Parameter 4: `uint[ ]`

Each packet’s start time, in µs, relative to the start of the layer.

Parameter 5: `uint[ ]`

Each packet’s duration, in µs.

Parameter 6: `double[ ]`

Each packet’s X demand position on the powder bed, in mm.

Parameter 7: `double[ ]`

Each packet’s Y demand position on the powder bed, in mm.

Parameter 8: `uint[ ]`

The averaged value of all samples collected by the LaserVIEW sensor during the duration of each packet.

Parameter 9: `uint[ ]`

The averaged value of all samples collected by the MeltVIEW plasma sensor during the duration of each packet.

Parameter 10: `uint[ ]`

The averaged value of all samples collected by the MeltVIEW meltpool sensor during the duration of each packet.

Return: `string`

The plug-in should return either:

- a string containing the word “Success”, if the method completed correctly, or
- a string containing as much detail as desired on the cause of failure, or
- a JSON string conforming to the Renishaw Central return API.

6.3.11.9 The `Shutdown` method

This method is called by DataHUB when all data for a build has been sent to the plug-in, or when the *Initialise* call to the plug-in failed. In this method the plug-in should perform any final processing on the build data, release any resources it may have acquired, and so on. The method has no parameters.

Parameters: None

Return: `string`

The plug-in should return a string containing either:

- the word “Success”, if the method has completed correctly, or
- as much detail as desired on the cause of failure.

If DataHUB receives a string that does not conform, the content of the returned string is added to the audit log for later diagnosis.

6.4 Plug-in API V2

This section is a guide to building, testing and deploying a V2.0 plug-in software component for DataHUB. The plug-in development cycle begins with installing DataHUB on the development PC, then uses Visual Studio to create a .NET assembly containing the plug-in code. With the appropriate Visual Studio debugging configuration, the plug-in can be tested and debugged to verify its correctness before its final deployment to a Data Collection PC (DCPC) in a production AM system.

In this section, the structure of a build *token stream* is defined. A code example then shows how to write a token stream processing plug-in using the V2.0 API. The second code example demonstrates the use of *filters* to process a token stream. Then, the implementation of the *ExportPackets* plug-in (as distributed with the DataHUB software) is recreated. The last code examples demonstrate how a plug-in's results may be sent to Renishaw Central to appear alongside other data collected by Renishaw Additive Manufacturing systems.

The section concludes with a definition of the V2.0 API.

6.4.1 Prerequisite activities

The following documents should be read in conjunction with this document:

- *InfiniAM® and DataHUB* Software installation guide (Renishaw part no. H-5800-4349)
- *DataHUB* user guide (Renishaw part no. H-5800-4761)
- Renishaw Licence Manager v2.x User Manual

Before proceeding, the following activities should be performed on the PC to be used for plug-in development:

- Ensure the PC is equipped with a GPU at least at the DataHUB minimum specification (see the data collection PC hardware specification in the *InfiniAM and DataHUB software* installation guide).
- Verify up-to-date drivers for the GPU are installed.
- Ensure the licence server has sufficient appropriate licences for DataHUB and Spectral API.
- Verify network access to the licence server.
- Ensure a version of Microsoft Visual Studio is installed that supports .NET Framework 4.7.2.

6.4.2 Token streams

6.4.2.1 Token stream structure

The DataHUB service transforms the process monitoring data of a build into a token stream to be consumed by V2.0 plug-ins. The order and type of tokens in a token stream forms a pseudo-hierarchical description of the build process monitoring data.

The tokens in a stream fall into two categories: paired start and end tokens, and tokens bounded by paired start and end tokens.

6.4.2.2 The stream for one laser, for one layer

The process monitoring data for one laser for one layer of a build is represented in the following token stream:

```
StartOfLayerLaserData (Layer, Laser)
   DataSourceInformation
   Sample
   Sample
   …
   Sample
EndOfLayerLaserData (Layer, Laser)
```

Note that the paired `StartOfLayerLaserData` and `EndOfLayerLaserData` tokens mark a conceptual boundary around the tokens specific to the laser.

The `DataSourceInformation` token contains values from the process monitoring data sources (i.e., DAT files) containing the sample data.

Each `Sample` token contains the process monitoring data values gathered by the LaserVIEW and MeltVIEW hardware for a single 100 KHz sample.

6.4.2.3 The stream for one layer

The set of all process monitoring data for a layer is combined into the token stream, repeating the above per-laser structure once for each laser of the machine:

```
StartOfLayerData (Layer)
   StartOfLayerLaserData (Layer, Laser)
   …
   EndOfLayerLaserData (Layer, Laser)
   StartOfLayerLaserData (Layer, Laser)
      …
   EndOfLayerLaserData (Layer, Laser)
   StartOfLayerLaserData (Layer, Laser)
      …
```

```
    EndOfLayerLaserData (Layer, Laser)
    StartOfLayerLaserData (Layer, Laser)
       …
    EndOfLayerLaserData (Layer, Laser)
EndOfLayerData (Layer)
```

Again, the layer-specific tokens are bounded by paired `StartOfLayerData` and `EndOfLayerData` tokens.

6.4.2.4 The stream for a build

The entire data for a build forms a token stream repeating the above per-layer structure for each layer:

```
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
```

6.4.2.5 Splits

A minor detail was omitted from the token stream structure just described, that is `Split` tokens. Split tokens are added to the token stream to bookmark points at which the data changes character. They are added as a generic signal, in addition to the more specific `StartOf/EndOf…` tokens. When processing a token stream, often it is simpler to act on this generic token rather than the specific markers.

6.4.2.6 Full definition of a build token stream

The full definition of a token stream for a build is:

```
StartOfLayerData (Layer)
   StartOfLayerLaserData (Layer, Laser)
      DataSourceInformation
      Split
      Sample
      Sample
      …
      Sample
      Split
   EndOfLayerLaserData (Layer, Laser)
   Split
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
```

6.4.2.7 Adding a token type

The API defines a base class `Token` from which all tokens are derived. To add a new token type into the token stream, derive it from this class:

```
public class NewTokenType : Token
{
}
```

A full catalogue of the API tokens can be found in Section 6.4, “Plug-in API V2”.

6.4.3 Installing DataHUB for plug-in development

The DataHUB service must be installed on the development PC. During installation, the option of which licences to claim is offered. Ensure the “LaserVIEW/MeltVIEW Pluggability” option is checked:

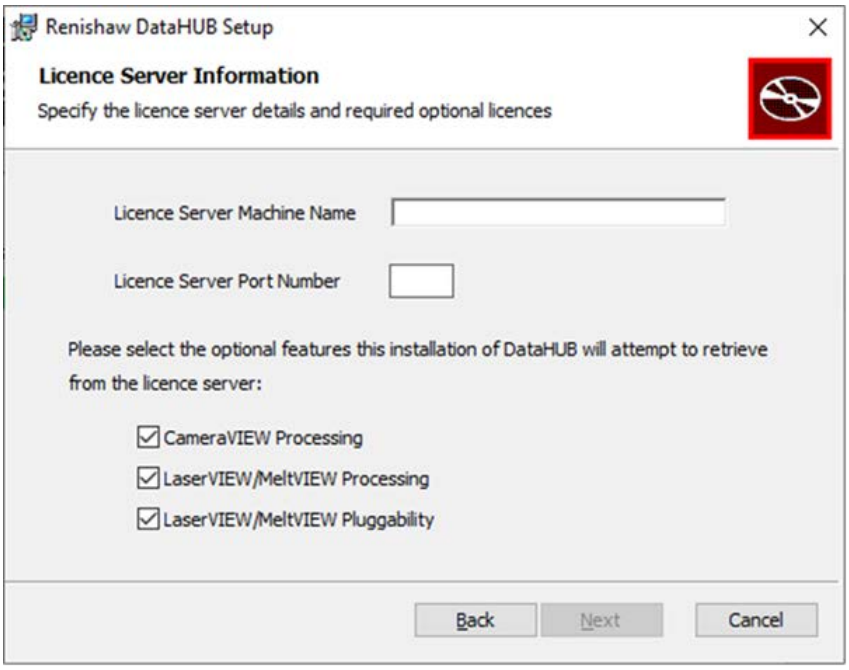


Figure 21 Licensing DataHUB

6.4.4 Creating a plug-in in Visual Studio

The following sections of this document demonstrate how to create a plug-in solution using Visual Studio.

Plug-ins are written in C# and define a public class implementing the specific set of public methods DataHUB uses to identify, validate and use instances of the plug-in when a build's process monitoring data is processed.

NOTE: While the workflow shown below uses Microsoft Visual Studio 2019, it is broadly similar in other Microsoft Visual Studio versions.

6.4.4.1 Creating the solution

Start Visual Studio and choose *Create a new project*. From the available options, choose *Class Library (.NET Framework)*:

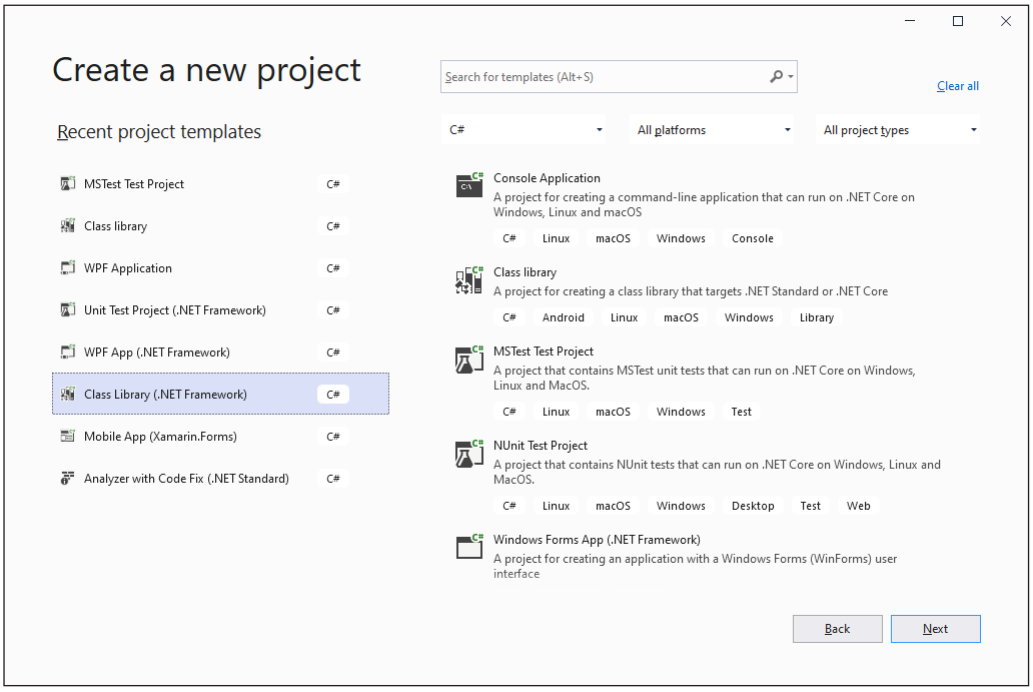


Figure 22 Creating the solution

Next, configure the project:

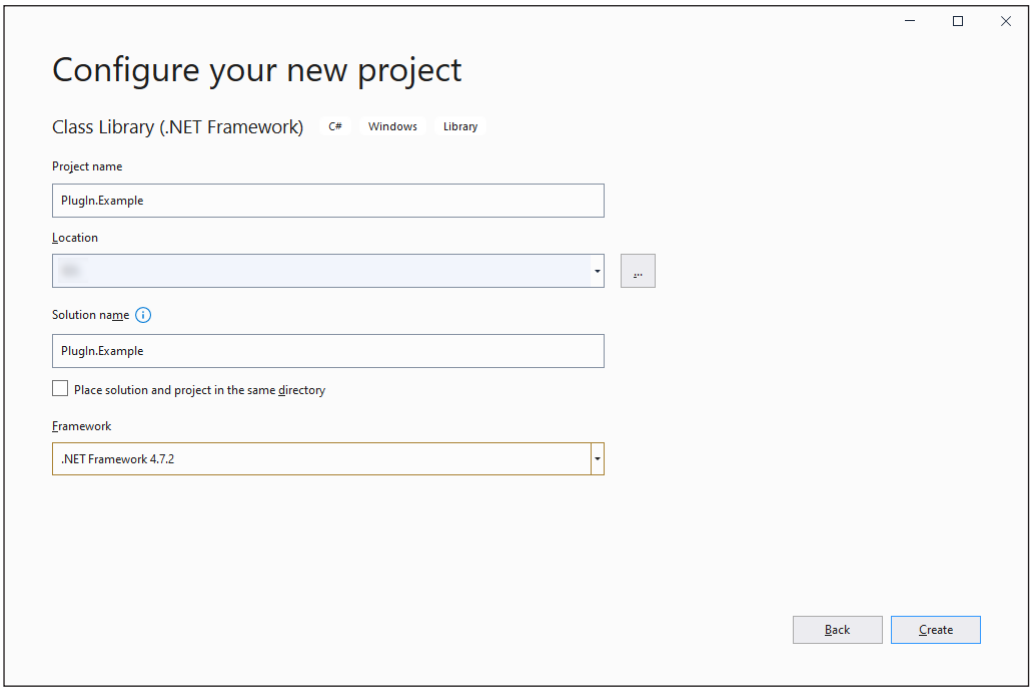


Figure 23 Configuring the project

DataHUB requires a plug-in assembly must start with **PlugIn.** (the word “PlugIn” followed by a full-stop). Select a folder of your choice as the project’s *Location* and create the solution.

NOTE: DataHUB supports plug-ins targeting x86/x64/Any CPU. It is recommended to target Framework 4.7.2 or higher.

6.4.4.2 Add a reference to the Plug-in API assembly

To develop a plug-in for the V2.0 API, a reference to the plug-in API assembly is required. Right-click the project “References” solution item, and chose “Add Reference…”

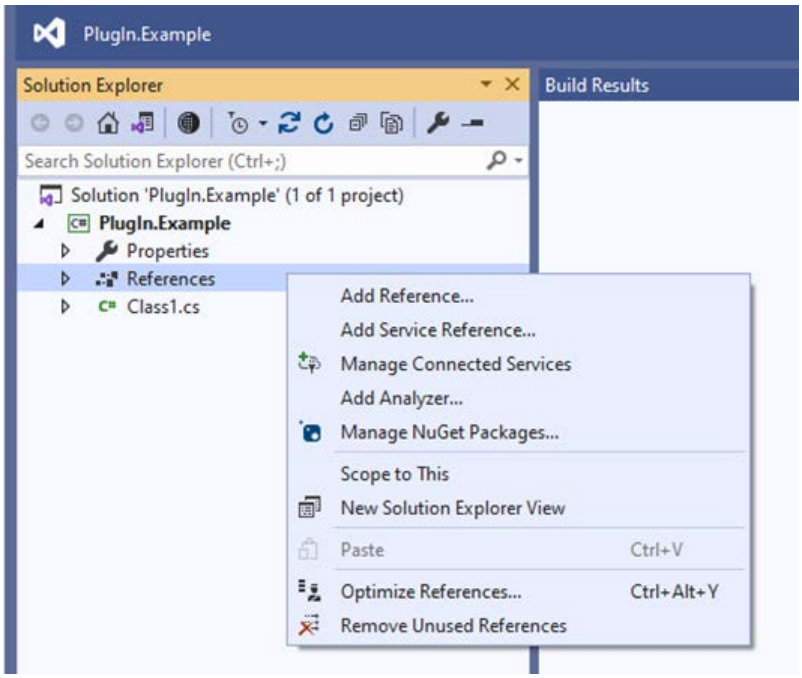


Figure 24 Add a reference

In the Reference Manager, opt to “Browse…”, and locate the “PlugIns.API.v2.dll” in the DataHUB installation folder (i.e., <$PROGRAMFILES$\Renishaw\DataHUB>):

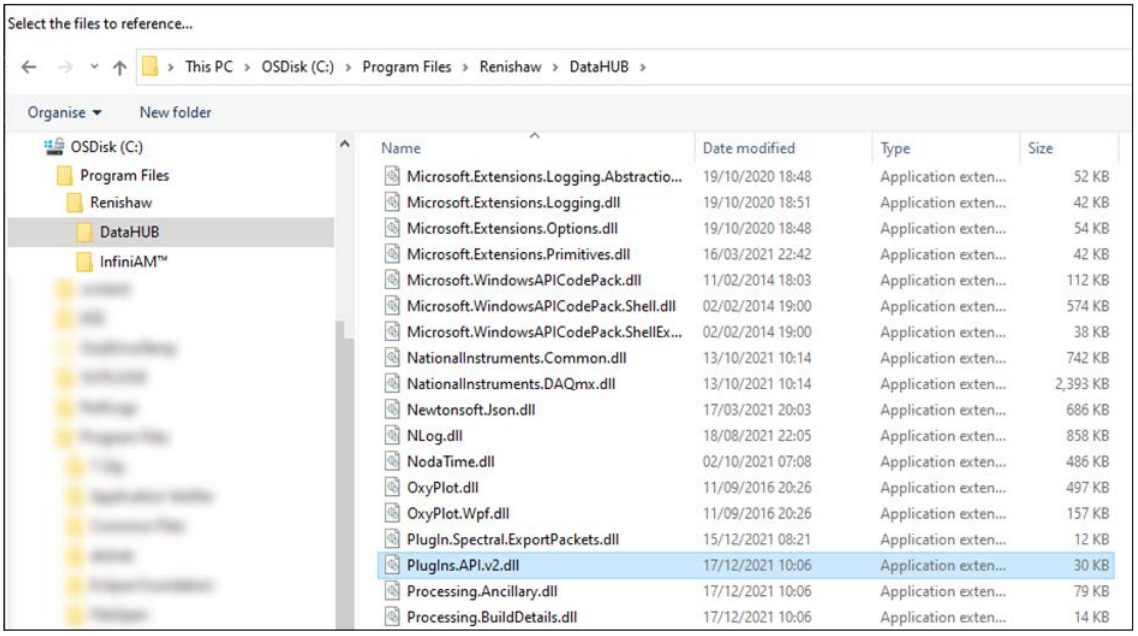


Figure 25 The API assembly

The assembly will then be listed in the dependencies of the plug-in project:

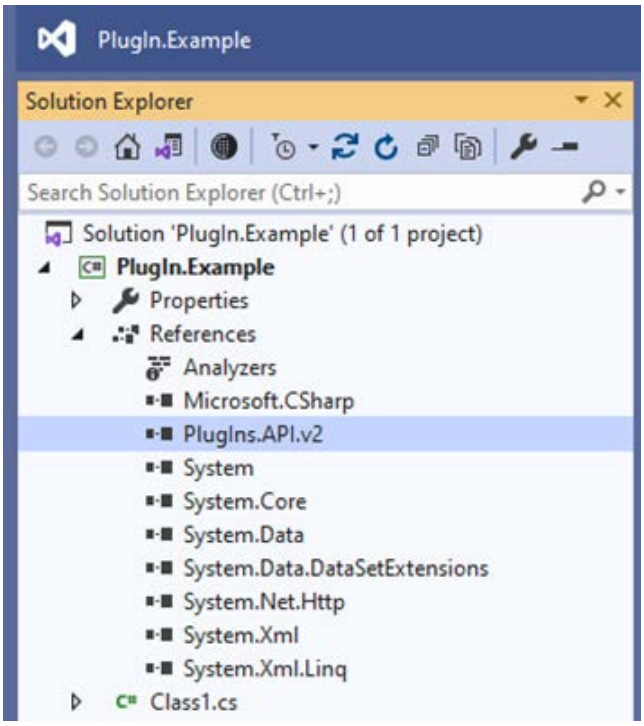


Figure 26 The plug-in API assembly referenced as a dependency

6.4.4.3 Naming the plug-in type

The single default source file Class1.cs contains the definition of Class1, and we will use this as the starting point.

```
namespace PlugIn.Example
{
    public class Class1
    {
    }
}
```

Once DataHUB discovers a potential plug-in assembly, it searches for a public type `PlugIn`, so rename `Class1` as `PlugIn`:

```
namespace PlugIn.Example
{
    public class PlugIn
    {
    }
}
```

6.4.4.4 Adding the API

With a potential plug-in type, DataHUB requires it implements the following public methods:

```
public PlugIn() (generated automatically)
public static string APIVersion()
public static string DisplayName()
public string Initialise(…)
public string Process(…)
public string Shutdown()
```

Add the following using statements and method implementations to the class:

```
using System.Collections.Generic;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
public class PlugIn
{
   public static string APIVersion() => “2”;
   public static string DisplayName() => “Example plug-in”;
   public string Initialise(string initialisationData) => InitialiseReturns.Success();
   public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
   public string Shutdown() => ShutdownReturns.Success();
}
```

Notice all API methods return a `string`. The content of the returned string is used to signal to DataHUB the result of the method. As each method is fleshed out, below, the role of this return value will be explained.

6.4.4.5 The static `APIVersion` method

This method reports to DataHUB the version of the API used by the plug-in. To use the API V2.0, this must return the literal “2”:

```
   public static string APIVersion() => “2”;
```

6.4.4.6 The static `DisplayName` method

The content of the string returned by this method is used as the plug-in name displayed in DataHUB Monitor, in creating the plug-in’s output folder, and so on. Although it is not mandatory for this name to be unique, making it so helps identify the specific plug-in when, for example, multiple versions are installed on a DCPC.

```
   public static string DisplayName() => “Example plug-in”;
```

6.4.4.7 The `Initialise` method

This method is called by DataHUB at the start of a build, before any layer data is sent to the plug-in. Hence it passes *build invariant* data to the plug-in. The minimal implementation is to return that the plug-in initialised without failure:

```
public string Initialise(string initialisationData) => InitialiseReturns.Success();
```

6.4.4.8 The `Process` method

This method is called by DataHUB when all data for a layer has been received. The minimal implementation is to return that the plug-in processed the data without failure:

```
public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
```

6.4.4.9 The `Shutdown` method

This method is called by DataHUB when all data for a build has been sent to the plug-in, or when any previous call to the plug-in failed. Once again, the minimal implementation is to return without failure:

```
public string Shutdown() => ShutdownReturns.Success();
```

6.4.4.10 Deploying the plug-in for debugging

Build the solution in *Debug* and open a file browser and locate the output folder which will contain a *.dll* file named as per the name given to the project, for example, *PlugIn.Example.dll*. (Ancillary files such as debug symbols, etc. may also be present.)

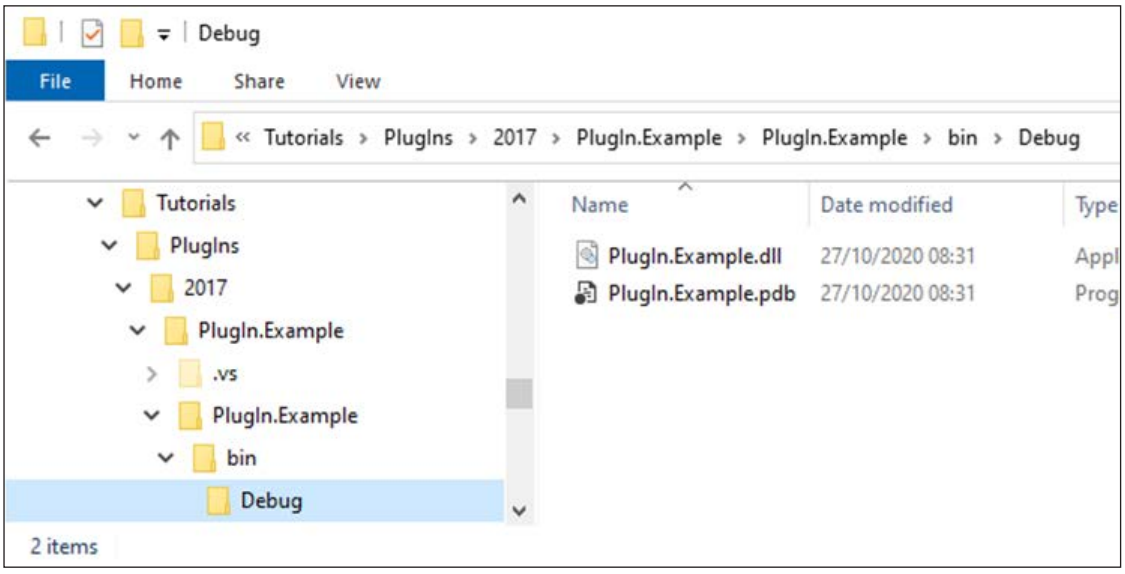


Figure 27 The plug-in binaries

Copy this folder and paste it into the “PlugIns” subfolder in the data folder of DataHUB i.e., <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>.

NOTE: Typically, $PROGRAMDATA$ will be located at <C:\ProgramData> but may be hidden, in which case the Windows File Explorer application option “View, Show/hide, Hidden items” option should be checked.

Administrator privileges may be required to modify the contents of this folder.

Rename the *Debug* folder to suit the name of the plug-in – in the image below, the folder has been named *PlugIn.Example*.

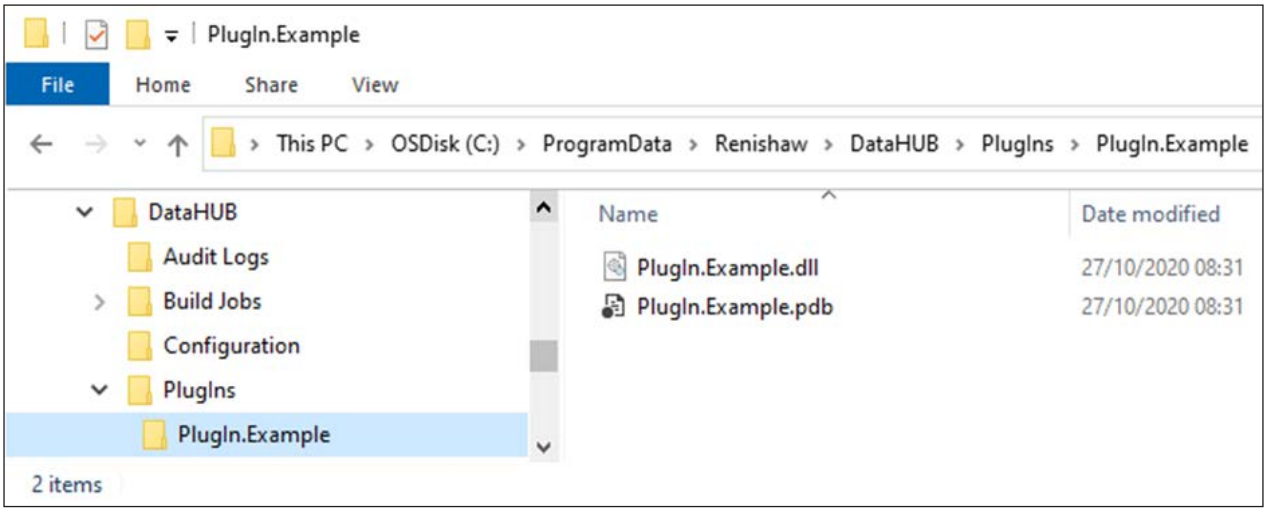


Figure 28 The plug-in deployed

6.4.4.11 Registering the plug-in

DataHUB service must be restarted to register the new plug-in. Use the *Services* application to do this, by right-clicking on *DataHUB Service* and choosing *Restart*:

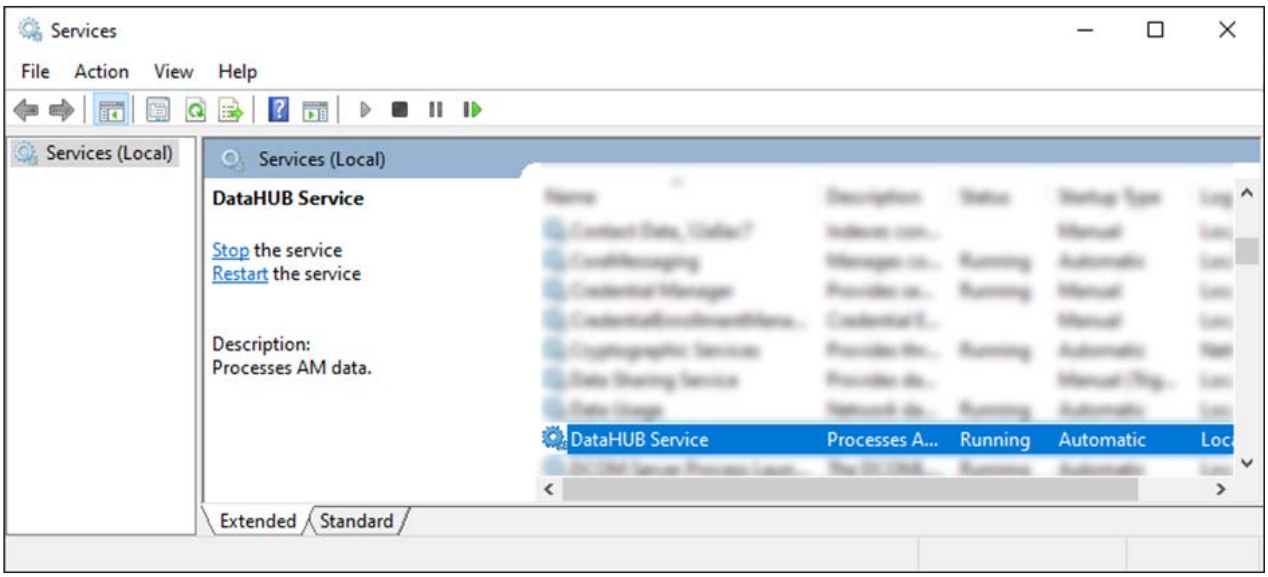


Figure 29 The Windows Service application

After a few seconds, the DataHUB service will show its status as *Running*.

6.4.4.12 Checking registration

DataHUB generates a log file as it runs, detailing important events such as build starts, build errors and, of course, plug-in auditing. Using File Explorer, locate the log folder <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>:

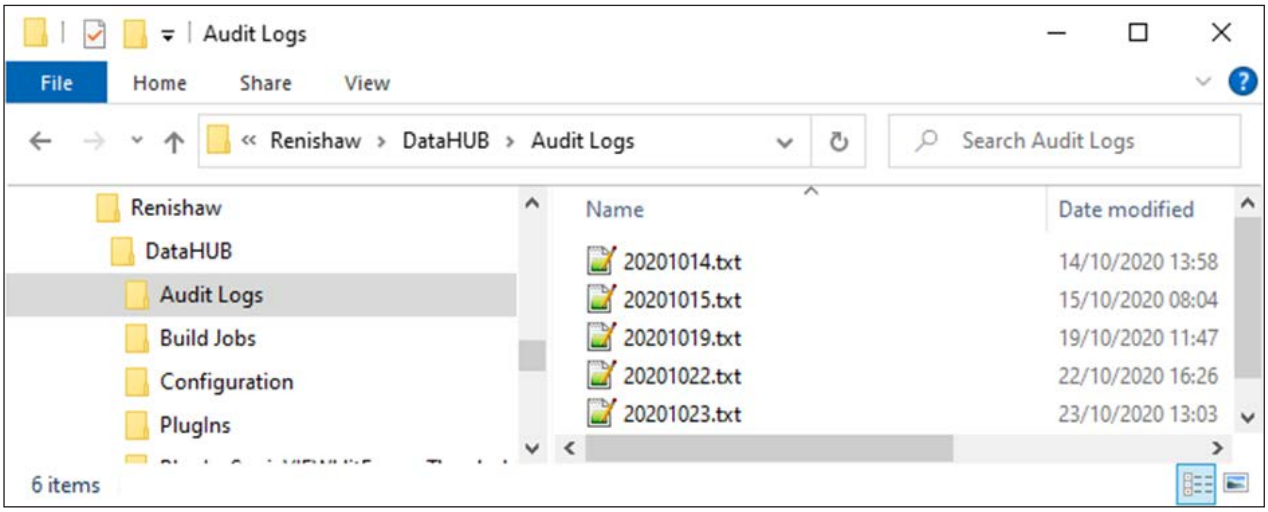


Figure 30 The DataHUB log folder

Open the most recent log file – it should have today's date – scroll to the end and look for lines detailing the discovery of valid plug-ins:

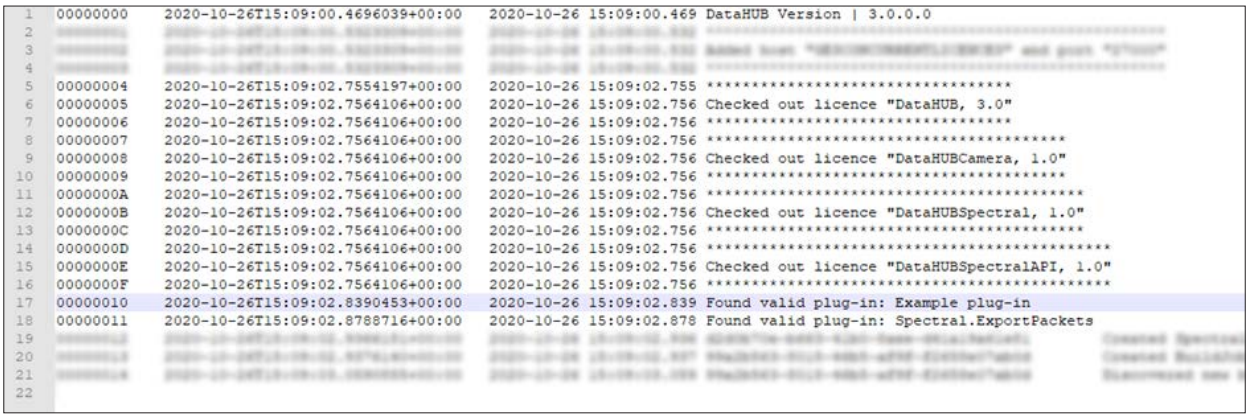


Figure 31 Valid plug-ins found by DataHUB

6.4.4.13 Timeouts

As DataHUB cannot guarantee every call to a plug-in instance will return in a timely fashion, it adopts a timeout policy. If a plug-in fails to complete within a certain duration, it is presumed faulty and shut down. The default timeouts are:

- For *Initialise*, 20 seconds
- For *Process* (shared with API V1.0's *ProcessPackets*), 200 seconds
- For *Shutdown*, 20 seconds

During debugging, it will often be the case that these timeouts will be exceeded, causing DataHUB to shut down the plug-in prematurely. To allow for more debugging time, the timeout durations can be increased by editing the file <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> in a suitable XML or text editor:

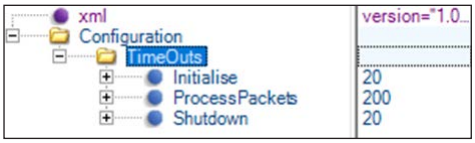


Figure 32 The default plug-in timeouts

NOTE: The DataHUB service must be restarted for timeout duration changes to take effect.

6.4.4.14 Debugging the plug-in

With a successfully deployed valid plug-in, it is now possible to debug its execution. The plug-in itself is not directly executable, but by attaching the Visual Studio debugger to the DataHUB service and starting a build job, the plug-in implementation will be invoked by the service and breakpoints set in the code will be hit.

NOTE: To debug via the service, Visual Studio may require administrator privileges. These can be granted by right-clicking the Visual Studio program shortcut and selecting "Run as administrator".

In Visual Studio, set breakpoints at the start of the *Initialise*, *Process* and *Shutdown* methods:

Figure 33 Breakpoints set in *Initialise* and *ProcessPackets*

Open the Debug menu and select “Attach to process…”:

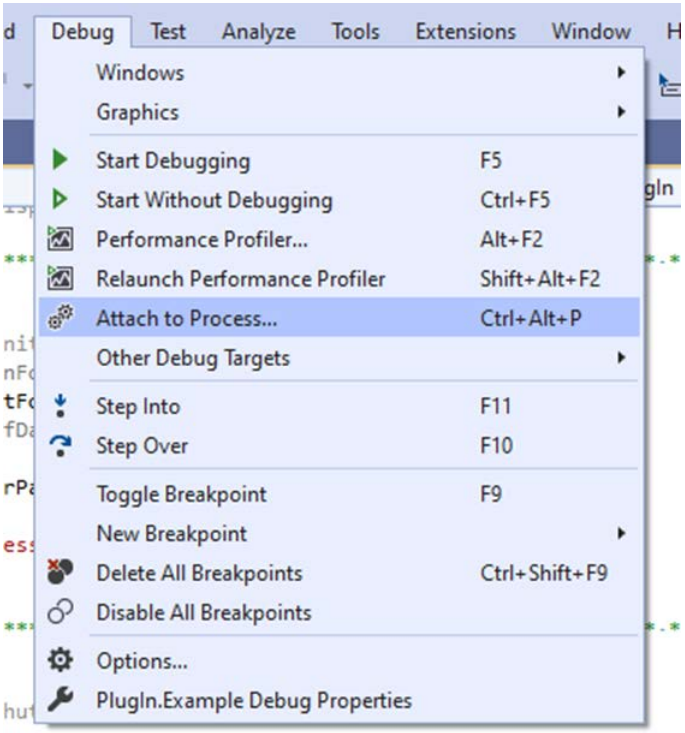


Figure 34 Attaching to a process for debugging

In the list of running processes, locate *DataHUB Service.exe* – you may need to check “Show processes from all users”:

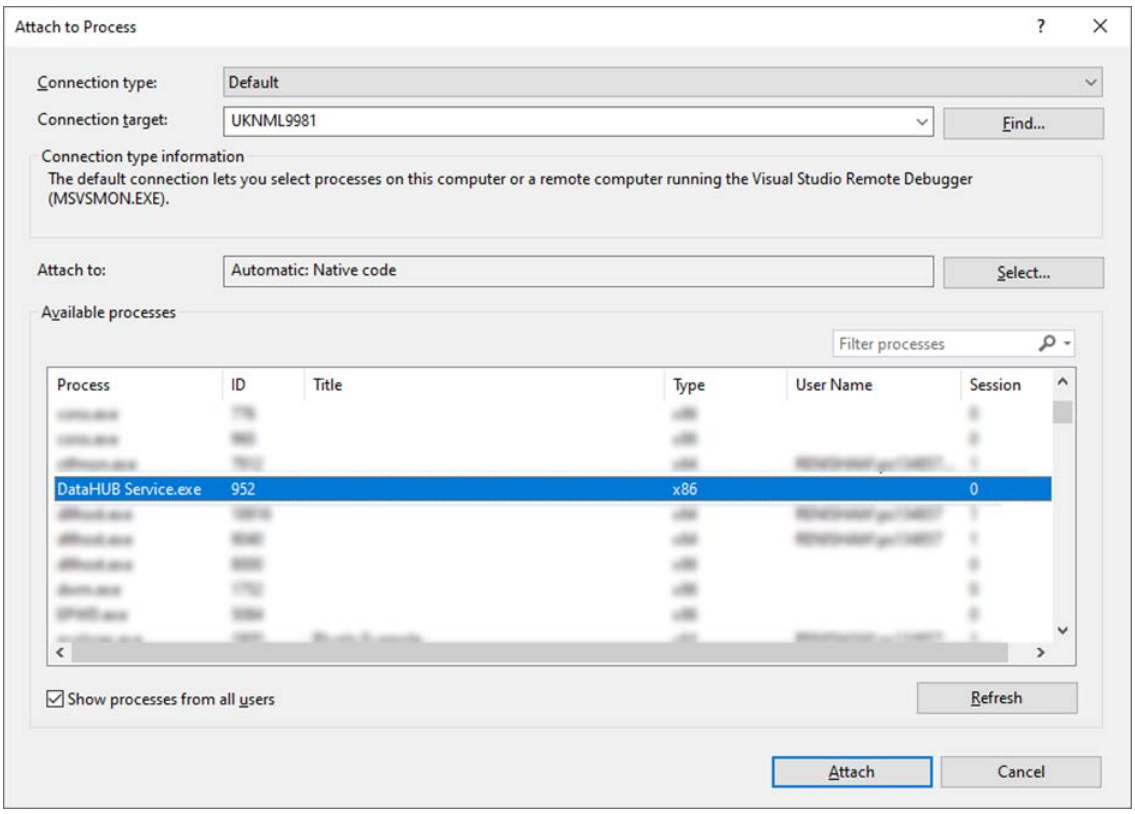


Figure 35 The DataHUB Service process

Press the “Attach” button, and Visual Studio will begin a debugging session against the DataHUB service.

For DataHUB to call the plug-in and hence for the breakpoints to be hit, a build must be started. This can be done via DataHUB Generator, or, if you have already generated some builds, simply by duplicating a build folder in <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs>.

DataHUB will automatically detect the new build and begin processing data. One of the service’s preliminary tasks is to create (by invoking the parameterless constructor) an instance of each registered plug-in. When DataHUB then calls the instance *Initialise* method, the first breakpoint will be hit. The values of the method parameter value will be those of the build in question, for example:

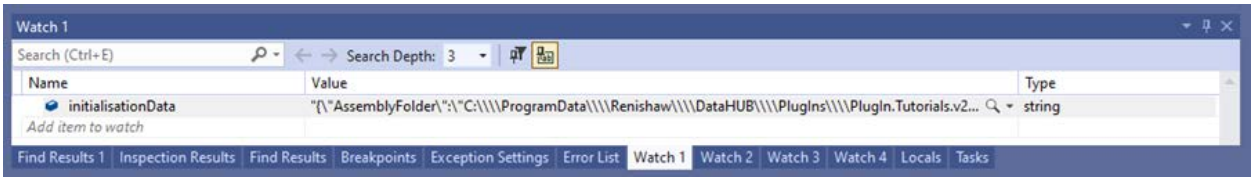


Figure 36 Initialise method parameter values

From then on, existing data (and, for a live build, new data as it arrives) is processed and the *Process* method of the instance is called with parameter values for the LaserVIEW and MeltVIEW data associated for a layer:

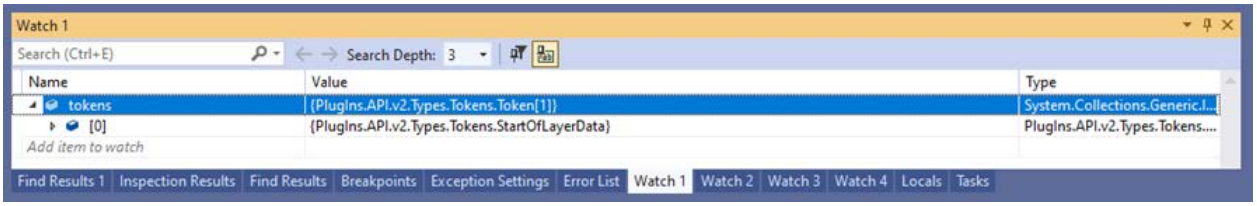


Figure 37 Process method parameter values

6.4.4.15 Deploying the plug-in for production

In preparation for deploying the plug-in to a production DCPC, it is critical that a *Release* build be compiled, locally deployed, and tested. Once the functionality of the plug-in is validated and verified, it may be deployed to production DCPCs.

The process for deployment to a production DCPC is very similar to that of deploying to a development PC. The *Release* folder should be copied to the *PlugIns* subfolder in the data folder of DataHUB (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) on the DCPC and renamed to suit the name of the plug-in. The DataHUB service then must be restarted to register the new plug-in: use the *Services* application to do this. The success or otherwise of registering the new plug-in is logged in the audit log.

NOTE: DataHUB is designed to resume all in-progress data processing tasks after a restart, so it is not necessary to wait for all in-progress tasks to complete before restarting. However, only new data processing tasks will use the newly registered plug-in.

6.4.4.16 Diagnosing plug-in failures in production

In the production environment, DataHUB Monitor will show (alongside other build processing tasks) the processing status for the plug-in:

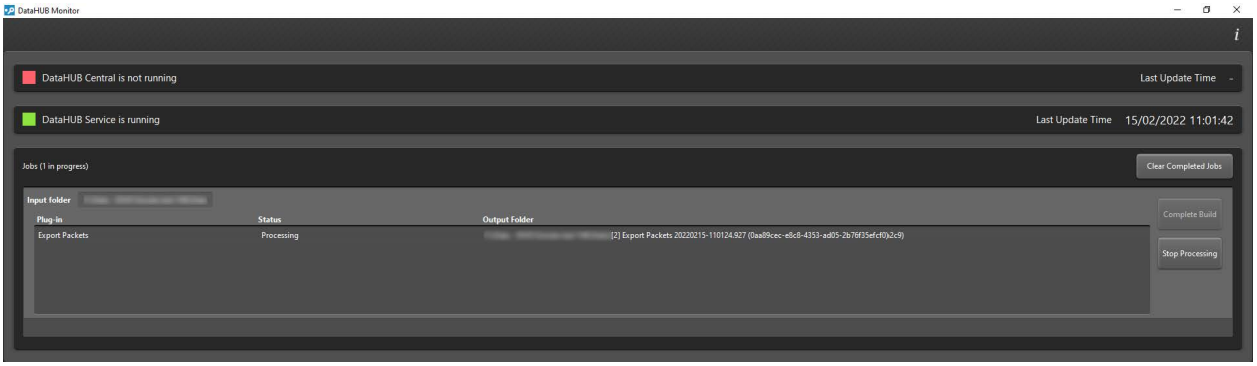


Figure 38 A build running with plug-ins

If the status of a plug-in is *Error*, the audit log will contain records of the failure, either as entries added by DataHUB itself (for example, the failure to load the plug-in assembly due to missing dependencies), or by the plug-in via the contents of the return value from the plug-in's *Initialise*, *Process* or *Shutdown* methods.

6.4.5 Example 1 – Processing a token stream

This first code example demonstrates how to process a token stream to extract minimum and maximum values for each channel of process monitoring data, for each laser, for each layer. Several key concepts will be covered:

- Decoding initialisation data
- Processing tokens
- The instance's output folder

6.4.5.1 Creation

The minimal plug-in implementation is as follows:

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example1
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 1";
      public string Initialise(string initialisationData) => InitialiseReturns.Success();
      public string Process(IreadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
   }
}
```

The API helper classes `InitialiseReturns`, `ProcessReturns`, and `ShutdownReturns` provide convenient methods for generating API return values; here, the Success methods return a standard "Success" string recognised by DataHUB.

6.4.5.2 Initialisation

When DataHUB initialises this plug-in, the first task is to store the location of the plug-in instance's unique output folder for later use when writing to file. The method decodes (using the Newtonsoft.Json library) the `initialisationData` string and extracts the output folder path:

```
public class PlugIn
{
   …
   public string Initialise(string initialisationData)
   {
      var initialisationValues = Jobject.Parse(initialisationData);
      _outputFolder = initialisationValues["OutputFolder"].Value<string>();
      return InitialiseReturns.Success();
   }
   private string _outputFolder;
   …
}
```

NOTE: Initialisation data is encoded as a JSON formatted string.

6.4.5.3 Processing the token stream

With initialisation successful, the plug-in instance will receive the token stream of the build. As some process monitoring data sets may be very large, the token stream is delivered in batches; each time `Process` is called, the next batch of stream tokens is received.

Handling for `Sample` tokens can now be added to the `Process` method:

```
…
public string Process(IreadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         case Sample sample:
            _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
            _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
            _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
            _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
            _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
            _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
            _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
```

```
            _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
            _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
            _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
            break;
      }
   return ProcessReturns.SuccessWithoutInformation();
}
private static (double, double) MinMaxForProperty((double Min, double Max) currentValue, double value)
{
   var (min, max) = currentValue;
   return (Math.Min(min, value), Math.Max(max, value));
}
private static (int, int) MinMaxForProperty((int Min, int Max) currentValue, int value)
{
   var (min, max) = currentValue;
   return (Math.Min(min, value), Math.Max(max, value));
}
private (double Min, double Max) _demandX             = (double.MaxValue, double.MinValue);
private (double Min, double Max) _demandY             = (double.MaxValue, double.MinValue);
private (double Min, double Max) _demandFocus         = (double.MaxValue, double.MinValue);
private (int Min, int Max)       _laserModulate       = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _demandLaserPower    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserVIEW           = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserOutputPower    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserBackReflection = (int.MaxValue, int.MinValue);
…
```

Several fields are initialised, which will store the running minimum and maximum for each of the process monitoring data properties of a `Sample` token.

Note that the return from `Process` has changed to `SuccessWithoutInformation()` which indicates to DataHUB that the plug-in has completed processing the token batch but does not want to add a message to the DataHUB logs; this change ensures that the DataHUB logs are not spammed with trivial “Success” messages.

NOTE: Token streams are batched; `Process` will be called multiple times.

NOTE: Not every call to `Process` needs to return a logged message to DataHUB service.

The value of each `Sample` token's process monitoring data properties is used to update the running minimum and maximum. Note that some properties have the type of double, and some integer.

NOTE: A `Sample` token has the following process monitoring data properties:

- DemandX (double)
- DemandY (double)
- DemandFocus (double)
- DemandLaserPower (intege
- LaserModulate (integer)
- MeltVIEWPlasma (integer)
- MeltVIEWMeltpool (integer)
- LaserVIEW (integer)
- LaserOutputPower (integer)
- LaserBackReflection (integer)

6.4.5.4 Writing to file

The final task of the plug-in is to write results to file when the plug-in processes an `EndOfLayerLaserData` token:

```
…
public string Process(IreadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            var outputFilePath = Path.Combine(_outputFolder, “Min & Max for “
                                                          + $”layer {layer}, “
                                                          + $”laser {laser}”
                                                          + “.txt”);
            File.AppendAllText(outputFilePath, FormatMinAndMax(“DemandX”,            _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax(“DemandY”,            _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax(“DemandFocus”,        _demandFocus));
```

```
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower",     _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",        _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",       _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool",     _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",            _laserVIEW));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower",     _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection", _laserBackReflection));
            _demandX              = (double.MaxValue, double.MinValue);
            _demandY              = (double.MaxValue, double.MinValue);
            _demandFocus          = (double.MaxValue, double.MinValue);
            _laserModulate        = (int.MaxValue, int.MinValue);
            _demandLaserPower     = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma       = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool     = (int.MaxValue, int.MinValue);
            _laserVIEW            = (int.MaxValue, int.MinValue);
            _laserOutputPower     = (int.MaxValue, int.MinValue);
            _laserBackReflection  = (int.MaxValue, int.MinValue);
            return ProcessReturns.Success($"Processed layer {layer}, laser {laser}");
        …
    }
}
private static string FormatMinAndMax(string propertyName, (double Min, double Max) propertyMinAndMax)
{
   return $"{propertyName}\t"
        + $"{propertyMinAndMax.Min:F3}\t"
        + $"{_ propertyMinAndMax.Max:F3}\t"
        + $"{Environment.NewLine}";
}
private static string FormatMinAndMax(string propertyName, (int Min, int Max) propertyMinAndMax)
{
   return $"{propertyName}\t"
        + $"{propertyMinAndMax.Min}\t"
        + $"{propertyMinAndMax.Max}\t"
        + $"{Environment.NewLine}";
}
…
```

The output path for the results file is built by combining the output folder cached during initialisation and the layer and laser numbers for the tokens just processed. The minimum and maximum values for each `Sample` property are written to the file, and then are reset ready for processing the tokens for the next layer.

The plug-in returns a “Success” result with a message that will be audited by DataHUB.

NOTE: DataHUB assigns to every instance of a plug-in run a unique output folder path, which should be used when persisting plug-in output.

6.4.5.5 Final implementation

The complete plug-in is as follows:

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Types.Tokens;
using PlugIns.API.v2.Helpers;
namespace PlugIn.Tutorials.v2.Example1
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial Example 1”;
      public string Initialise(string initialisationData)
      {
         var initialisationValues = Jobject.Parse(initialisationData);
         _outputFolder = initialisationValues[“OutputFolder”].Value<string>();
         return InitialiseReturns.Success();
      }
      public string Process(IreadOnlyCollection<Token> tokens)
      {
         foreach (var token in tokens)
            switch (token)
            {
               case Sample sample:
                  _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
                  _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
                  _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
                  _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
                  _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
```

```
            _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
            _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
            _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
            _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
            _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
          break;
        case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            var outputFilePath = Path.Combine(_outputFolder, "Min & Max for "
                                                           + $"layer {layer}, "
                                                           + $"laser {laser}"
                                                           + ".txt");
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandX",          _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandY",          _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",      _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",    _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower",
                                                               _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",
                                                               _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool",
                                                               _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",
                                                               _laserVIEW
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower",
                                                               _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection",
                                                               _laserBackReflection));
            _demandX             = (double.MaxValue, double.MinValue);
            _demandY             = (double.MaxValue, double.MinValue);
            _demandFocus         = (double.MaxValue, double.MinValue);
            _laserModulate       = (int.MaxValue, int.MinValue);
            _demandLaserPower    = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
            _laserVIEW           = (int.MaxValue, int.MinValue);
            _laserOutputPower    = (int.MaxValue, int.MinValue);
            _laserBackReflection = (int.MaxValue, int.MinValue);
            return ProcessReturns.Success($"Processed layer {layer}, laser {laser}");
```

```
        }
    return ProcessReturns.SuccessWithoutInformation();
}
public string Shutdown() => ShutdownReturns.Success();
private string _outputFolder;
private (double Min, double Max) _demandX             = (double.MaxValue, double.MinValue);
private (double Min, double Max) _demandY             = (double.MaxValue, double.MinValue);
private (double Min, double Max) _demandFocus         = (double.MaxValue, double.MinValue);
private (int Min, int Max)       _laserModulate       = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _demandLaserPower    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserVIEW           = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserOutputPower    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserBackReflection = (int.MaxValue, int.MinValue);

private static string FormatMinAndMax(string propertyName,
                                      (double Min, double Max) propertyMinAndMax)
{
    return $"{propertyName}\t"
         + $"{propertyMinAndMax.Min:F3}\t"
         + $"{propertyMinAndMax.Max:F3}\t"
         + $"{Environment.NewLine}";
}
private static string FormatMinAndMax(string propertyName,
                                      (int Min, int Max) propertyMinAndMax)
{
    return $"{propertyName}\t"
         + $"{propertyMinAndMax.Min}\t"
         + $"{propertyMinAndMax.Max}\t"
         + $"{Environment.NewLine}";
}
private static (double, double) MinMaxForProperty((double Min, double Max) currentValue,
                                                  double value)
{
    var (min, max) = currentValue;
    return (Math.Min(min, value), Math.Max(max, value));
}
private static (int, int) MinMaxForProperty((int Min, int Max) currentValue,
                                            int value)
```

```
      {
         var (min, max) = currentValue;
         return (Math.Min(min, value), Math.Max(max, value));
      }
   }
}
```

6.4.6 Filters

A *filter* in the API is a unit of code that processes one input token and outputs zero or more tokens. A filter might, after appropriate processing, output the input token, output new tokens, or even swallow the input token. By these methods, a filter transforms an input token stream into a new output token stream.

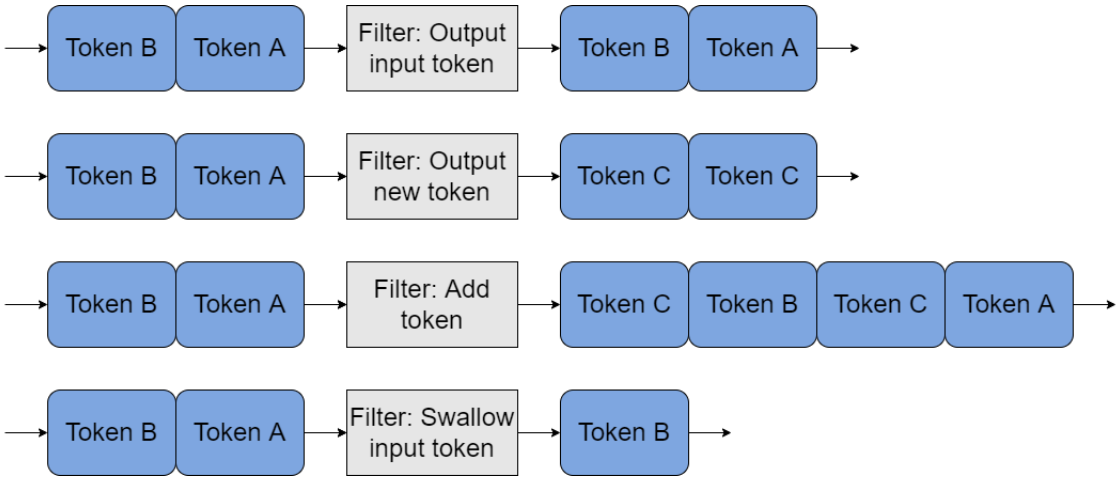


Figure 39 Methods of filter output

6.4.6.1 Pipelines

Given two filters, the API provides a method of combining them into a *pipeline*. The fact that a pipeline has the same signature as a single filter means that pipelines and filters may be combined into longer, more complex pipelines. When a pipeline processes a token stream, the output of the first filter forms the input to the second, and so on until the output of the last filter forms the final output of the pipeline.

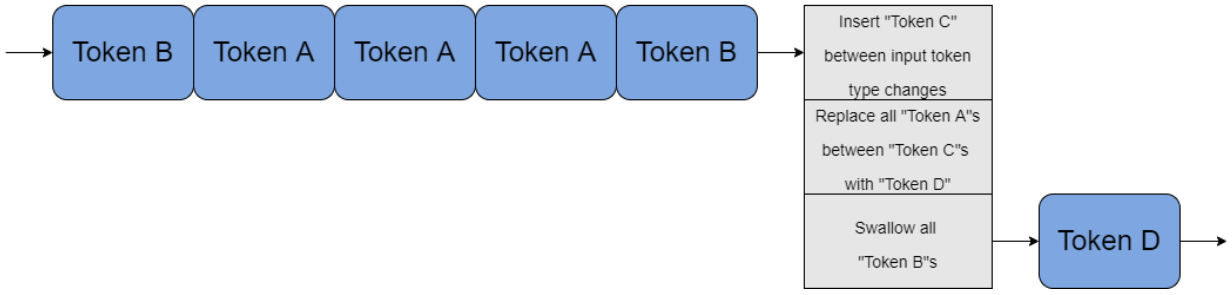


Figure 40 A pipeline transforming a token stream

6.4.6.2 Summarising samples into packets using a filter pipeline

This section illustrates the transformation of an input token stream of samples, via several filter stages, into an output token stream of summary data.

The series of filters forming the pipeline are:

- Split when Laser Modulate changes
- Emit if laser is firing
- Split when DemandX changes
- Split when DemandY changes
- Collate into packets of samples
- Summarise packets of samples

6.4.6.2.1 Input token stream

The input token stream consists of seven sample tokens:

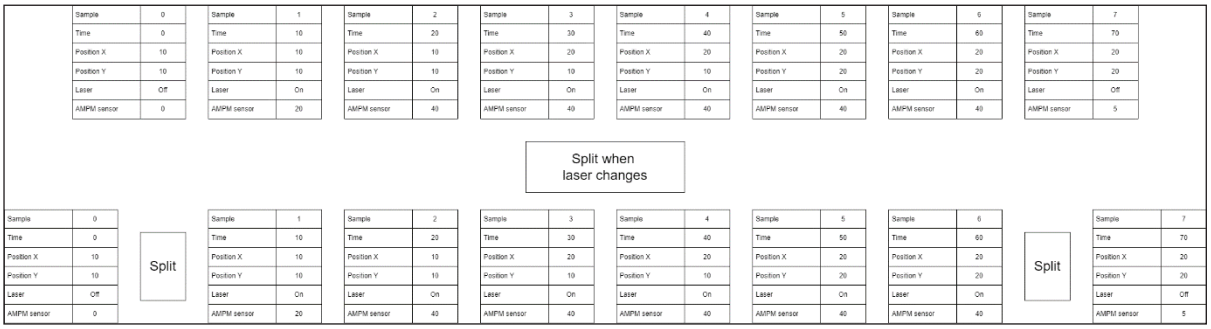
Sample	0	Sample	1	Sample	2	Sample	3	Sample	4	Sample	5	Sample	6	Sample	7
Time	0	Time	10	Time	20	Time	30	Time	40	Time	50	Time	60	Time	70
Position X	10	Position X	10	Position X	10	Position X	20	Position X	20	Position X	20	Position X	20	Position X	20
Position Y	10	Position Y	10	Position Y	10	Position Y	10	Position Y	10	Position Y	20	Position Y	20	Position Y	20
Laser	Off	Laser	On	Laser	On	Laser	On	Laser	On	Laser	On	Laser	On	Laser	Off
AMPM sensor	0	AMPM sensor	20	AMPM sensor	40	AMPM sensor	40	AMPM sensor	40	AMPM sensor	40	AMPM sensor	40	AMPM sensor	5

The operation of the laser can be traced through the various values in each sample:

- The laser begins at position (10, 10) at time 0, and is not firing
- The laser fires at time 20
- The laser moves to position (20, 10) at time 30
- The laser moves to position (20, 20) at time 50
- The laser stops firing at time 70

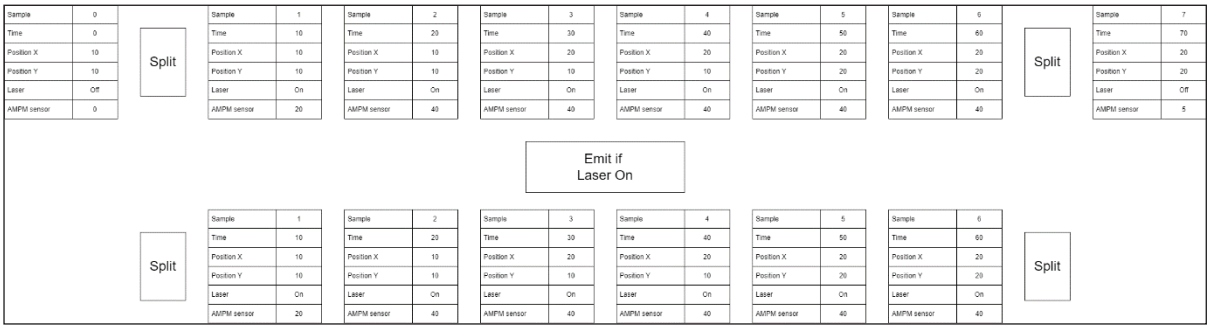
6.4.6.2.2 Filter 1 – Split when LaserModulate changes

The first filter inserts `Split` tokens between changes in the laser modulate channel:



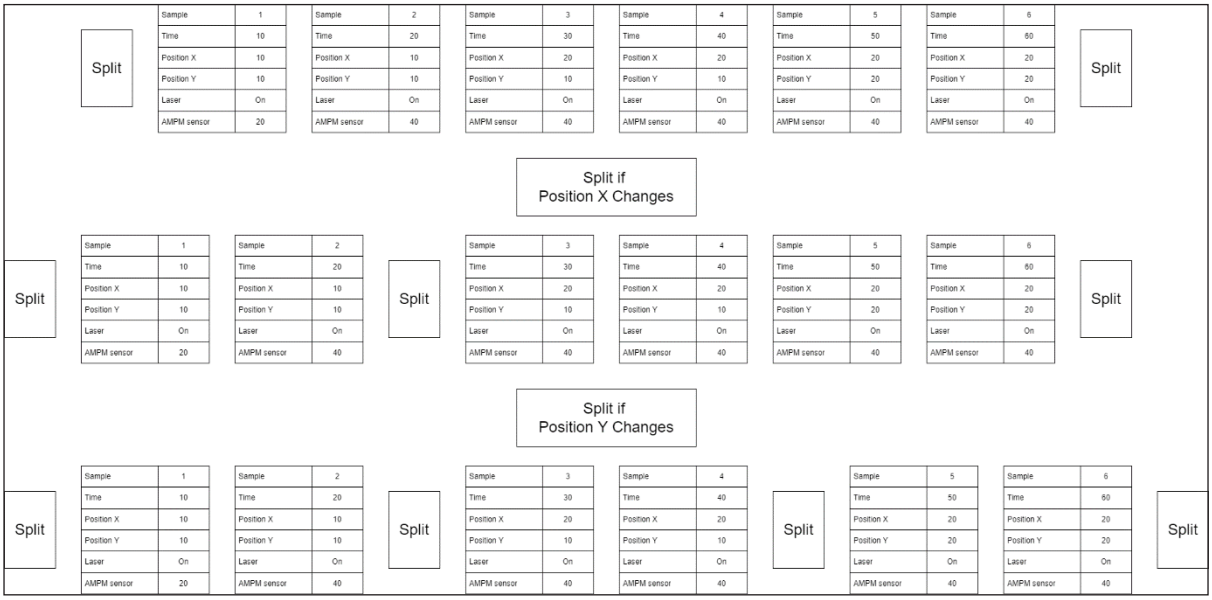
6.4.6.2.3 Filter 2 – Emit if laser on

The next filter strips out any samples where the laser is off:



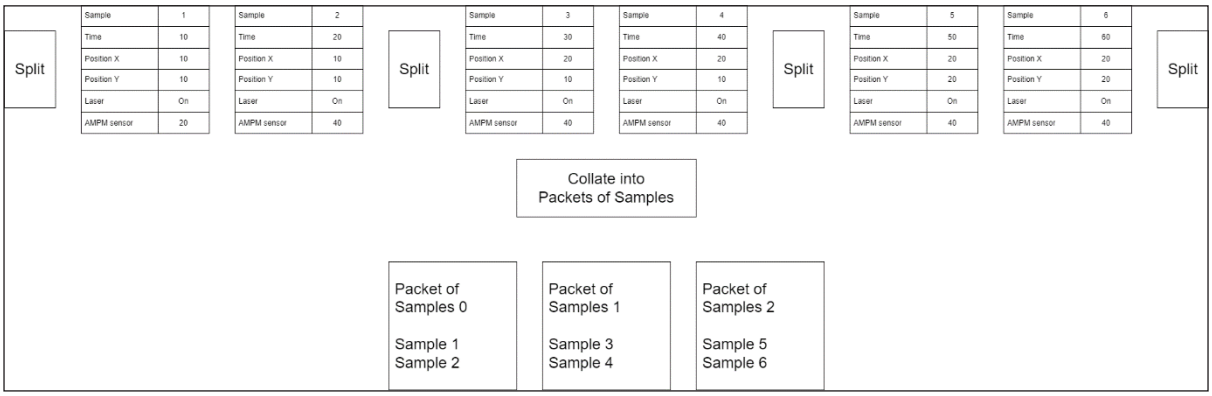
6.4.6.2.4 Filter 3 and 4 – Split when Position changes

The next two filters insert `Split` tokens between samples where the laser position changes:



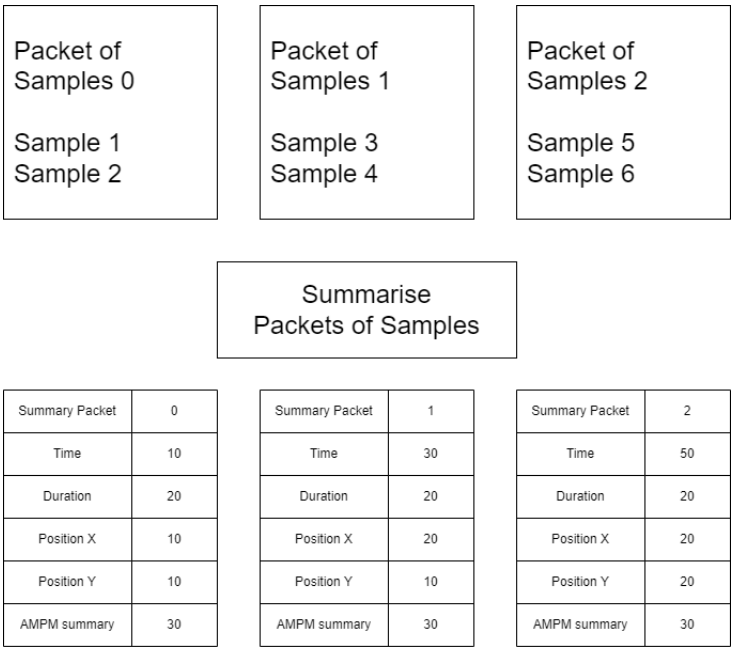
6.4.6.2.5 Filter 5 – Collate into packets of samples

The samples are now bounded by `Split` tokens, between each of which are the samples where the laser was firing and stationary – i.e., the samples for a single meltpool. These groups of samples are then collated.



6.4.6.2.6 Filter 6 – Summarised packets of samples

Finally, the collated samples are summarised:



Each of these final summary packet tokens contains values describing the start time (the time of the first contributing sample), duration (the number of contributing samples × the duration of a single sample), position and summary process monitoring values.

6.4.7 Example 2 – Filters

This example extends the minimum/maximum plug-in using filters provided in the API. These key concepts will be covered:

- Filters
- Processing tokens through a filter

6.4.7.1 Filters

Filters process and modify a token stream by adding, removing, forwarding or changing tokens. A filter processes a single token and outputs zero or more tokens. This one-to-many mapping provides a powerful mechanism for the modification of a token stream as it is processed.

In many process monitoring analysis tasks, only samples recorded when the laser was firing are of interest (as they contain data about the meltpool). Filtering out samples recorded while the laser was not firing also reduces the number of tokens that must be considered in the analysis. The built-in filter `EmitSampleIf.LaserModulate.IsOn` is designed for this purpose. The plug-in can pass each `Sample` token through this filter to remove all laser-off samples, thereby producing minimum and maximum values relevant to meltpool formation.

In C#, a delegate is a type that represents a method with a particular signature. A delegate's value is assigned with a *method instance*, and that method may be invoked via the delegate.

A new namespace `PlugIn.Tutorials.v2.Example2` and plug-in class is added with a delegate `FilterOutLaserOffSamples` that is assigned the filter `EmitSampleIf.LaserModulate.IsOn()` when the plug-in instance is created:

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example2
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 2";
      public string Initialise(string initialisationData)
      {
         var initialisationValues = JObject.Parse(initialisationData);
         _outputFolder = initialisationValues["OutputFolder"].Value<string>();
         return InitialiseReturns.Success();
      }
```

```
    public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
    public string Shutdown() => ShutdownReturns.Success();
    private string _outputFolder;
    private Func<Token, IEnumerable<Token>> FilterOutLaserOffSamples { get; }
        = EmitSampleIf.LaserModulate.IsOn();
  }
}
```

The Process method is implemented next, using an LINQ expression to pass each token batch first through the filter (via the delegate) and then through a method that processes the resulting modified token stream:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
  var messages = tokens.FilteredBy(FilterOutLaserOffSamples).
                        CollateMessagesFrom(ProcessTokens);
  return ProcessReturns.Success(messages);
}
private IEnumerable<string > ProcessTokens(IEnumerable<Token> tokens)
{
  foreach (var token in tokens)
    switch (token)
    {
      case Sample sample:
        _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
        _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
        _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
        _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
        _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
        _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
        _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
        _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
        _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
        _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
        break;
      case EndOfLayerLaserData endOfLayerLaserData:
        var layer = endOfLayerLaserData.Layer.Value;
        var laser = endOfLayerLaserData.Laser.Value;
        var outputFilePath = Path.Combine(_outputFolder, “Min & Max for “
                                                       + $”layer {layer}, “
                                                       + $”laser {laser}”
```

```
                                                    + ".txt");
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandX",             _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandY",             _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",         _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",       _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower",    _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",      _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool",    _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",           _laserVIEW));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower",    _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection", _laserBackReflection));
            _demandX             = (double.MaxValue, double.MinValue);
            _demandY             = (double.MaxValue, double.MinValue);
            _demandFocus         = (double.MaxValue, double.MinValue);
            _laserModulate       = (int.MaxValue, int.MinValue);
            _demandLaserPower    = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
            _laserVIEW           = (int.MaxValue, int.MinValue);
            _laserOutputPower    = (int.MaxValue, int.MinValue);
            _laserBackReflection  = (int.MaxValue, int.MinValue);
            yield return "Processed layer {layer}, laser {laser}";
        }
}
```

NOTE: The various fields (`_demandX`, etc.) and helper methods (`FormatMinAndMax`, etc.) are omitted for brevity and are implemented as in the first example.

The LINQ expression comprises two stages. First, each batch of tokens is passed to the `FilterOutLaserOffTokens` filter via an API helper `FilteredBy`. Then, the tokens in the resulting token stream (where `Sample` tokens will be only those samples recorded when the laser was firing) are passed to `ProcessToken` which performs the processing particular to each token type of interest, i.e., `Sample` and `EndOfLayerLaserData`. The resulting strings returned by `ProcessTokens` are collected and returned to DataHUB by the `Success` helper.

NOTE: C#'s LINQ syntax is designed for processing streams of data.

NOTE: Token streams may be modified through a filter.

NOTE: A filter returns zero or more tokens for each input token.

The plug-in will now populate the results files with minimum and maximum channel values for samples recorded while the laser was firing.

6.4.7.2 Final implementation

The complete plug-in is as follows:

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example2
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 2";
      public string Initialise(string initialisationData)
      {
         var initialisationValues = JObject.Parse(initialisationData);
         _outputFolder = initialisationValues["OutputFolder"].Value<string>();
         return InitialiseReturns.Success();
      }
      public string Process(IReadOnlyCollection<Token> tokens)
      {
         var messages = tokens.FilteredBy(FilterOutLaserOffSamples).
                               CollateMessagesFrom(ProcessTokens);
         return ProcessReturns.Success(messages);
      }
      private IEnumerable<string> ProcessTokens(IEnumerable<Token> tokens)
      {
        foreach (var token in tokens)
         switch (token)
         {
            case Sample sample:
               _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
               _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
               _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
               _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
```

```
            _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
            _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
            _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
            _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
            _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
            _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
            break;
        case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            var outputFilePath = Path.Combine(_outputFolder, "Min & Max for "
                                                           + $"layer {layer}, "
                                                           + $"laser {laser}"
                                                           + ".txt");
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandX",          _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandY",          _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",      _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",    _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower", _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",   _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool", _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",        _laserVIEW));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower", _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection",
                                                               _laserBackReflection));
            _demandX             = (double.MaxValue, double.MinValue);
            _demandY             = (double.MaxValue, double.MinValue);
            _demandFocus         = (double.MaxValue, double.MinValue);
            _laserModulate       = (int.MaxValue, int.MinValue);
            _demandLaserPower    = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
            _laserVIEW           = (int.MaxValue, int.MinValue);
            _laserOutputPower    = (int.MaxValue, int.MinValue);
            _laserBackReflection = (int.MaxValue, int.MinValue);
            yield return $"Processed layer {layer}, laser {laser}";
            break;
    }
}
public string Shutdown() => ShutdownReturns.Success();
```

```
    private string _outputFolder;
    private Func<Token, IEnumerable<Token>> FilterOutLaserOffSamples { get; }
       = EmitSampleIf.LaserModulate.IsOn();
    private (double Min, double Max) _demandX             = (double.MaxValue, double.MinValue);
    private (double Min, double Max) _demandY             = (double.MaxValue, double.MinValue);
    private (double Min, double Max) _demandFocus         = (double.MaxValue, double.MinValue);
    private (int Min, int Max)       _laserModulate       = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _demandLaserPower    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserVIEW           = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserOutputPower    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserBackReflection = (int.MaxValue, int.MinValue);
    private static string FormatMinAndMax(string propertyName, (double Min, double Max) propertyMinAndMax)
    {
       return $"{propertyName}\t"
            + $"{propertyMinAndMax.Min:F3}\t"
            + $"{propertyMinAndMax.Max:F3}\t"
            + $"{Environment.NewLine}";
    }
    private static string FormatMinAndMax(string propertyName, (int Min, int Max) propertyMinAndMax)
    {
       return $"{propertyName}\t"
            + $"{propertyMinAndMax.Min}\t"
            + $"{propertyMinAndMax.Max}\t"
            + $"{Environment.NewLine}";
    }
    private static (double, double) MinMaxForProperty((double Min, double Max) currentValue, double value)
    {
       var (min, max) = currentValue;
       return (Math.Min(min, value), Math.Max(max, value));
    }
    private static (int, int) MinMaxForProperty((int Min, int Max) currentValue, int value)
    {
       var (min, max) = currentValue;
       return (Math.Min(min, value), Math.Max(max, value));
    }
  }
}
```

6.4.8 Example 3 – ExportPackets

While individual filters are useful, their full power is realised by combining them into pipelines (see “Pipelines” on page 6-56). A pipeline performs a complex operation on a token stream through a series of simple, reusable filters. The `ExportPackets` plug-in distributed with the DataHUB service does just that, and this example will recreate that implementation, covering:

- Combining filters into a pipeline.
- Processing a token stream through a pipeline.

6.4.8.1 Combining filters

The plug-in API provides two helpers `First` and `Then` to assist combining filters into pipelines. A pipeline is created like this:

```
First(<filter>).Then(<filter>).Then(<filter>)…Then(<filter>);
```

Any number of filters may be chained together, and the resulting pipeline behaves as a filter, mapping a single input token to zero or more output tokens.

The *ExportPackets* plug-in distributed with DataHUB processes a token stream into packets, where a packet is a summary of the process monitoring data collected for consecutive samples while a laser was firing at a particular demand X and Y position. To create packets from samples, a token stream is processed as follows:

1. Split the stream each time the laser on/off state changes.
2. Remove all laser-off samples.
3. Split the stream each time the X position changes.
4. Split the stream each time the Y position changes.
5. Gather each split-bounded set of samples as a packet of samples.
6. Produce mean and median summary values for each packet of samples.

NOTE: The stream’s laser-off samples is not filtered out first, as might at first seem sensible, because doing so would create erroneous packets should the laser remain at the same demand X and Y while turning on and off repeatedly.

The starting code for the plug-in is:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Text;
```

```
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
using static PlugIns.API.v2.Helpers.PositionHelpers;
using static PlugIns.API.v2.Helpers.Pipeline;
namespace PlugIn.Tutorials.v2.Example3
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial 3 - Export Packets";
      public string Initialise(string initialisationData)
      {
          _outputFolder = JObject.Parse(initialisationData)["OutputFolder"].Value<string>();
          SetColumnHeadings(_builder);
          return InitialiseReturns.Success();
      }
      public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
      private static void SetColumnHeadings(StringBuilder builder)
      {
         builder.Clear();
         builder.AppendLine("Start time\tDuration\t"
                            + "Demand X\t"
                            + "Demand Y\t"
                            + "Demand focus\t"
                            + "Demand laser power (mean)\t"
                            + "MeltVIEW plasma (mean)\t"
                            + "MeltVIEW melt pool (mean)\t"
                            + "LaserVIEW (mean)\t"
                            + "Laser back reflection (mean)\t"
                            + "Laser output power (mean)\t"
                            + "Demand laser power (median)\t"
                            + "MeltVIEW plasma (median)\t"
                            + "MeltVIEW melt pool (median)LaserVIEW (median)\t"
                            + "Laser back reflection (median)\t"
                            + "Laser output power (median)\t");
      }
      private readonly StringBuilder _builder = new StringBuilder();
```

```
      private string _outputFolder;
   }
}
```

The plug-in uses a `StringBuilder` to accumulate the summaries of all the sample packets; first it adds the headings for each column in the csv file.

The pipeline discussed above is created and assigned to the delegate `CollateIntoPackets`:

```
…
private Func<Token, IEnumerable<Token>> CollateIntoPackets { get; }
   = First(SplitWhen.LaserModulateChanges()).
     Then(EmitSampleIf.LaserModulate.IsOn()).
     Then(SplitWhen.DemandXChanges()).
     Then(SplitWhen.DemandYChanges()).
     Then(CollateInto.PacketOfSamples()).
     Then(SummarisePacketOfSamples.Summarise(new SummarisePacketOfSamples.SummaryMethod(
                                               "Mean",
                                               values => values.Average(x => x)),
                                             new SummarisePacketOfSamples.SummaryMethod(
                                               "Median",
                                               Median)));
private static double Median(IReadOnlyList<double> values)
{
   var halfwayIndex = values.Count >> 1;
   return values.Count % 2 == 1
            ? values[halfwayIndex]
            : (values[halfwayIndex] + values[halfwayIndex - 1]) * 0.5;
}
…
```

The pipeline consists of several built-in filters chained so that the final output from the pipeline will be a series of `SummaryPacket` tokens containing mean and median values. See “Summarising samples into packets using a filter pipeline” on page 6-57 for an overview of the pipeline’s function.

NOTE: Simple filters chained together form a pipeline capable of performing a complex transformation of an input token stream.

The final filter, `SummarisePacketOfSamples.Summarise`, is initialised with `SummaryMethods` which name the summary – “Mean” and “Median” – and provide the method by which its value is to be calculated. This is a flexible way of configuring how packets of samples will be summarised: many summary methods may be provided, for example, to produce means, medians, minimums and maximums:

```
SummarisePacketOfSamples.Summarise(
   new SummarisePacketOfSamples.SummaryMethod(“Mean”,    values => values.Average(x => x)),
   new SummarisePacketOfSamples.SummaryMethod(“Median”,  Median)),
   new SummarisePacketOfSamples.SummaryMethod(“Minimum”, values => values.Min(x => x)),
   new SummarisePacketOfSamples.SummaryMethod(“Maximum”, values => values.Max(x => x)))
```

The final step is to implement the *Process* method which will add (using the helper method `AddSummaryLine`) to the `StringBuilder` a line for each `SummaryPacket` token, and write the contents of the `StringBuilder` to the output file:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   var messages = tokens.FilteredBy(CollateIntoPackets).
                         CollateMessagesFrom(ProcessToken);
   return ProcessReturns.Success(messages);
}
private IEnumerable<string> ProcessToken(IEnumerable<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         case SummaryPacket summaryPacket:
            AddSummaryLine(_builder,
                           summaryPacket.Time,
                           summaryPacket.Duration,
                           summaryPacket.Summary[“Mean”],
                           summaryPacket.Summary[“Median”]);
         break;
         case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            File.AppendAllText(Path.Combine(_outputFolder,
                               $”Packet data for layer {layer}, laser {laser}.txt”), _builder.ToString());
            SetColumnHeadings(_builder);
            yield return $”Processed layer {layer}, laser {laser}”;
            break;
```

```
        }
}
private void AddSummaryLine(StringBuilder builder,
                            uint time,
                            uint duration,
                            SummaryPacket.SummaryValues means,
                            SummaryPacket.SummaryValues medians)
{
   builder.AppendLine($"{time}\t"
                    + $"{duration}\t"
                    + $"{FormatTo3dp(RoundToMicrons(means.DemandX,     1))}\t"
                    + $"{FormatTo3dp(RoundToMicrons(means.DemandY,     1))}\t"
                    + $"{FormatTo3dp(RoundToMicrons(means.DemandFocus, 1))}\t"
                    + $"{FormatTo3dp(means.DemandLaserPower)}\t"
                    + $"{FormatTo3dp(means.MeltVIEWPlasma)}\t"
                    + $"{FormatTo3dp(means.MeltVIEWMeltpool)}\t"
                    + $"{FormatTo3dp(means.LaserVIEW)}\t"
                    + $"{FormatTo3dp(means.LaserBackReflection)}\t"
                    + $"{FormatTo3dp(means.LaserOutputPower)}\t"
                    + $"{FormatTo3dp(medians.DemandLaserPower)}\t"
                    + $"{FormatTo3dp(medians.MeltVIEWPlasma)}\t"
                    + $"{FormatTo3dp(medians.MeltVIEWMeltpool)}\t"
                    + $"{FormatTo3dp(medians.LaserVIEW)}\t"
                    + $"{FormatTo3dp(medians.LaserBackReflection)}\t"
                    + $"{FormatTo3dp(medians.LaserOutputPower)}\t");
}
private string FormatTo3dp(double meansDemandLaserPower)
{
   return meansDemandLaserPower.ToString("F3", _currentCulture);
}
…
```

Each `SummaryPacket` token is added as a line of text, with formatting appropriate to each value, using the `StringBuilder`. When an `EndOfLayerLaserData` token is encountered, the plug-in writes the collated packet data to a file in the output folder.

NOTE: The build token stream can be processing through a pre-defined pipeline.

6.4.8.2 Final implementation

The complete plug-in is as follows:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Text;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
using static PlugIns.API.v2.Helpers.PositionHelpers;
using static PlugIns.API.v2.Helpers.Pipeline;
namespace PlugIn.Tutorials.v2.Example3
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial 3 - Export Packets”;
      public string Initialise(string initialisationData)
      {
         _outputFolder = JObject.Parse(initialisationData)[“OutputFolder”].Value<string>();
         SetColumnHeadings(_builder);
         return InitialiseReturns.Success();
      }
      public string Process(IReadOnlyCollection<Token> tokens)
      {
         var messages = tokens.FilteredBy(CollateIntoPackets).
                              CollateMessagesFrom(ProcessToken);
         return ProcessReturns.Success(messages);
      }
      public string Shutdown() => ShutdownReturns.Success();
      private IEnumerable<string> ProcessToken(IEnumerable<Token> tokens)
      {
         foreach (var token in tokens)
            switch (token)
            {
               case SummaryPacket summaryPacket:
```

```
                AddSummaryLine(_builder,
                               summaryPacket.Time,
                               summaryPacket.Duration,
                               summaryPacket.Summary[“Mean”],
                               summaryPacket.Summary[“Median”]);
                break;
            case EndOfLayerLaserData endOfLayerLaserData:
                var layer = endOfLayerLaserData.Layer.Value;
                var laser = endOfLayerLaserData.Laser.Value;
                File.AppendAllText(Path.Combine(_outputFolder,
                                                $”Packet data for layer {layer}, laser {laser}.txt”),
                                   _builder.ToString());
                SetColumnHeadings(_builder);
                yield return $”Processed layer {layer}, laser {laser}”;
                break;
        }
    }
    private static void SetColumnHeadings(StringBuilder builder)
    {
        builder.Clear();
        builder.AppendLine(“Start time\tDuration\t”
                           + “Demand X\t”
                           + “Demand Y\t”
                           + “Demand focus\t”
                           + “Demand laser power (mean)\t”
                           + “MeltVIEW plasma (mean)\t”
                           + “MeltVIEW melt pool (mean)\t”
                           + “LaserVIEW (mean)\t”
                           + “Laser back reflection (mean)\t”
                           + “Laser output power (mean)\t”
                           + “Demand laser power (median)\t”
                           + “MeltVIEW plasma (median)\t”
                           + “MeltVIEW melt pool (median)LaserVIEW (median)\t”
                           + “Laser back reflection (median)\t”
                           + “Laser output power (median)\t”);
    }
    private void AddSummaryLine(
        StringBuilder builder,
        uint time,
        uint duration,
```

```
    SummaryPacket.SummaryValues means,
    SummaryPacket.SummaryValues medians)
{
    builder.AppendLine($"{time}\t"
                     + $"{duration}\t"
                     + $"{FormatTo3dp(RoundToMicrons(means.DemandX,     1))}\t"
                     + $"{FormatTo3dp(RoundToMicrons(means.DemandY,     1))}\t"
                     + $"{FormatTo3dp(RoundToMicrons(means.DemandFocus, 1))}\t"
                     + $"{FormatTo3dp(means.DemandLaserPower)}\t"
                     + $"{FormatTo3dp(means.MeltVIEWPlasma)}\t"
                     + $"{FormatTo3dp(means.MeltVIEWMeltpool)}\t"
                     + $"{FormatTo3dp(means.LaserVIEW)}\t"
                     + $"{FormatTo3dp(means.LaserBackReflection)}\t"
                     + $"{FormatTo3dp(means.LaserOutputPower)}\t"
                     + $"{FormatTo3dp(medians.DemandLaserPower)}\t"
                     + $"{FormatTo3dp(medians.MeltVIEWPlasma)}\t"
                     + $"{FormatTo3dp(medians.MeltVIEWMeltpool)}\t"
                     + $"{FormatTo3dp(medians.LaserVIEW)}\t"
                     + $"{FormatTo3dp(medians.LaserBackReflection)}\t"
                     + $"{FormatTo3dp(medians.LaserOutputPower)}\t");
}
private Func<Token, IEnumerable<Token>> CollateIntoPackets { get; }
    = First(SplitWhen.LaserModulateChanges()).
      Then(EmitSampleIf.LaserModulate.IsOn()).
      Then(SplitWhen.DemandXChanges()).
      Then(SplitWhen.DemandYChanges()).
      Then(CollateInto.PacketOfSamples()).
      Then(SummarisePacketOfSamples.Summarise(new SummarisePacketOfSamples.SummaryMethod(
                                                "Mean",
                                                values => values.Average(x => x)),
                                              new SummarisePacketOfSamples.SummaryMethod(
                                                "Median",
                                                Median)));
private static double Median(IReadOnlyList<double> values)
{
    var halfwayValue = values.Count >> 1;
    return values.Count % 2 == 1
        ? values[halfwayValue]
        : (values[halfwayValue] + values[halfwayValue - 1]) * 0.5;
}
```

```
        private string FormatTo3dp(double meansDemandLaserPower)
        {
            return meansDemandLaserPower.ToString("F3", _currentCulture);
        }
        private readonly IFormatProvider _currentCulture = CultureInfo.CurrentCulture;
        private readonly StringBuilder _builder = new StringBuilder();
        private string _outputFolder;
    }
}
```

6.4.9 Example 4 – Returning time-series data to Renishaw Central

The following example demonstrates how a plug-in creates and populates a Renishaw Central time series with data points. The Renishaw Central web UI displays the time-series data as a graph.

The example plug-in defines a time series and for each layer adds a data point for the maximum value of the LaserVIEW channel. The time-series definition and its data points are returned to DataHUB via the return values of the *Initialise* and Process methods.

The key concepts that will be covered:

- Creating a Renishaw Central time series
- Adding data to a Renishaw Central time series

Beginning with the standard empty plug-in:

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Types.Tokens;
using PlugIns.API.v2.Types;
namespace PlugIn.Tutorials.v2.Example4
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial Example 4”;
      public string Initialise(string initialisationData) => InitialiseReturns.Success();
      public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
   }
}
```

6.4.9.1 Initialisation

When DataHUB initialises this plug-in, the task is to return a definition for the “LaserVIEW Max” time series. A time-series definition requires a name, unit type, and unit. Additional properties may also be defined, such as the Grouping property which categorises the time series into a grouping used by the Renishaw Central web frontend to collate and manage the display of related time series.

The time-series definition is returned to DataHUB using the `InitialiseReturns` method `SuccessAndCreateTimeSeries()`. DataHUB converts this returned value into posts to Renishaw Central with the necessary data to create the time series ready to be populated with data points:

```
…
public string Initialise(string initialisationData)
{
   _laserViewMax = TimeSeries.Factory().
                               Name(“LaserVIEW Max”)UnitType(“Intensity”).
                               DimensionlessUnit().
                               Grouping(“LaserVIEW”).
                               Create();
   return InitialiseReturns.SuccessAndCreateTimeSeries(_laserViewMax);
}
private TimeSeries _laserViewMax;
…
```

NOTE: One or more time-series definitions may be returned from the plug-in’s Initialise method.

NOTE: DataHUB manages the detail of posting data to Renishaw Central so plug-ins can concentrate simply on time-series definitions.

6.4.9.2 Processing

When processing each `Sample` token in the token stream, the current maximum value is compared to the sample's LaserVIEW channel value and updated if necessary:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
   {
      switch (token)
      {
         case Sample sample:
            if (sample.LaserVIEW >= _max.value)
               _max = (sample.Time, sample.LaserVIEW);
         break;
      }
   }
   return ProcessReturns.SuccessWithoutInformation();
}
private (uint time, int value) _max = (0, int.MinValue);
…
```

At the end of the laser data for the given layer, when the plugin receives the `EndOfLayerLaserData` token, a success message can be returned to DataHUB to signify that the data for the given layer and laser has been processed:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerLaserData endOfLayerLaserData:
            return ProcessReturns.Success($"Processed layer {endOfLayerLaserData.Layer.Value}, " +
                                          $"laser {endOfLayerLaserData.Laser.Value}");
         …
      }
   return ProcessReturns.SuccessWithoutInformation();
}
…
```

At the end of the layer data, the plug-in will receive an `EndOfLayerData` token. The `TimeSeries` object is used to produce an `UpdateTimeSeries` object with a time-series value containing the layer's final maximum value. The `ProcessReturns` helpers provide a method `SuccessAndSendData()` which takes an array of `UpdateTimeSeries` objects.

The cached maximum value for the layer is reset ready for the next batch of tokens, and the time-series update returned:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   …
   case EndOfLayerData endOfLayerData:
      var laserViewMaxUpdate = _laserViewMax.Update().
                                             WithValues((_max.Time, _max.Value)).
                                             NoLimits();
      _max = (0, int.MinValue);
      return ProcessReturns.SuccessAndSendData($"Processed layer {endOfLayerData.Layer.Value}",
                                               new[] {laserViewMaxUpdate});
   …
}
```

6.4.9.3 Final implementation

The complete plug-in is as follows:

```
using System.Collections.Generic;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example4
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 4";
      public string Initialise(string initialisationData)
      {
         _laserViewMax = TimeSeries.Factory().
                                    Name("LaserVIEW Max").
                                    UnitType("Intensity").
                                    DimensionlessUnit().
                                    Grouping("LaserVIEW").
```

```
                                    Create();
        return InitialiseReturns.SuccessAndCreateTimeSeries(_laserViewMax);
    }
    public string Process(IReadOnlyCollection<Token> tokens)
    {
        foreach (var token in tokens)
            switch (token)
            {
                case Sample sample:
                    if (sample.LaserVIEW >= _max.Value)
                        _max = (sample.Time, sample.LaserVIEW);
                    break;
                case EndOfLayerLaserData endOfLayerLaserData:
                    return ProcessReturns.Success($"Processed layer {endOfLayerLaserData.Layer.Value}, "
                        + $"laser {endOfLayerLaserData.Laser.Value}");
                case EndOfLayerData endOfLayerData:
                    var laserViewMaxUpdate = _laserViewMax.Update().
                                                           WithValues((_max.Time, _max.Value)).
                                                           NoLimits();
                    _max = (0, int.MinValue);
                    return ProcessReturns.SuccessAndSendData($"Processed layer {endOfLayerData.Layer.Value}",
                                                             new[] {laserViewMaxUpdate});
            }
        return ProcessReturns.SuccessWithoutInformation();
    }
    public string Shutdown() => ShutdownReturns.Success();
    private (uint Time, int Value) _max = (0, int.MinValue);
    private TimeSeries             _laserViewMax;
  }
}
```

6.4.10 Example 5 – Raising alerts on Renishaw Central

This example demonstrates how to raise alerts on Renishaw Central when certain values exceed pre-defined thresholds.

The key concepts that will be covered:

- Decoding initialisation data
- Raising alerts

6.4.10.1 Configuring the plug-in job with initialisation data

The thresholds beyond which this example will raise alerts is controlled through the plug-in initialisation data. In this example, the initialisation data matching the following structure is entered in the “Specify Plug-in Initialisation Data” stage of DataHUB Generator.

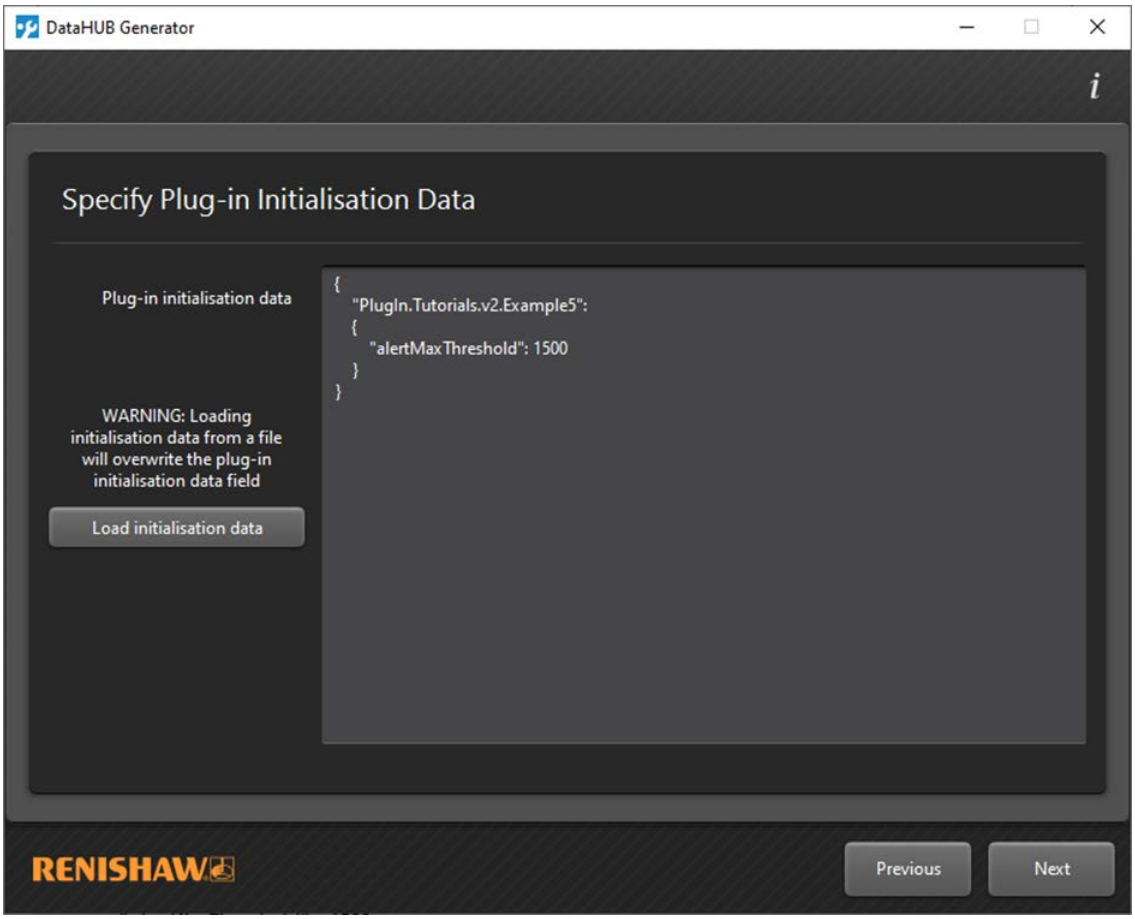


Figure 41 Plug-in initialisation data entry in DataHUB Generator

```
{
  “PlugIn.Tutorials.v2.Example5”:
  {
    “alertMaxThreshold”: 1500
  }
}
```

Beginning with the standard empty plug-in:

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example5
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial Example 5”;
      public string Initialise(string initialisationData) => InitialiseReturns.Success();
      public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
   }
}
```

6.4.10.2 Initialisation

On initialisation, the plug-in specific initialisation data can be accessed from within the “initialisationData” JSON object with the name that was provided on configuration of the job. In this example, “PlugIn.Tutorials.v2.Example5”.

```
…
public string Initialise(string initialisationData)
{
   var iv               = JObject.Parse(initialisationData);
   var id               = JObject.Parse(iv[“InitialisationData”].Value<string>());
   var example5InitData = id[“PlugIn.Tutorials.v2.Example5”];
   _alertMaxThreshold = example5InitData[“alertMaxThreshold”].Value<int>();
   InitialiseReturns.Success();
}
private int _alertMaxThreshold;
…
```

NOTE: The plug-in initialisation string specified when setting up the processing task in DataHUB Generator is available to the plug-in during initialisation.

6.4.10.3 Processing

The purpose of this plug-in is to raise an alert in Renishaw Central should the LaserVIEW maximum value for the layer exceed the defined threshold. Firstly, the current maximum value for the layer and associated sample time is cached and initialised. Upon processing of the sample tokens, the “LaserVIEW” channel is compared against the current maximum value, overwriting it if necessary. The sample time is also stored with the maximum value:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         case Sample sample:
            if (sample.LaserVIEW >= _max.value)
               _max = (sample.Time, sample.LaserVIEW);
            break;
      }
   return SuccessWithoutInformation();
}
private (uint time, int value) _max = (0, int.MinValue);
…
```

At the end of the laser data for the given layer, the plug-in will receive the `EndOfLayerLaserData` token. A success message can be returned to DataHUB to signify that the data for the given layer/laser has been processed successfully:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerLaserData endOfLayerLaserData:
            return Success($”Processed layer {endOfLayerLaserData.Layer}, “ +
                           $”laser {endOfLayerLaserData.Laser}”);
         …
      }
   return SuccessWithoutInformation();
}
…
```

At the end of all the samples for the layer, the plug-in receives an `EndOfLayerData` token, signifying there will be no more data for the given layer. Upon encountering this token, processing can be finalised and the results sent to Renishaw Central.

If the current maximum has exceeded the defined threshold, an alert is be created and added to the alerts list. The alert is created with a description, a severity level, a name and a time stamp.

The `ProcessReturns` helpers provide a method `SuccessAndSendData` which takes a message and an array of alerts and encodes that data to be sent back to DataHUB:

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerData endOfLayerData:
            var alerts = new List<Alert>();
            if (_max.value > _alertMaxThreshold)
               alerts.Add(Alert.Factory().
                                 Description(“LaserVIEW intensity has exceeded the maximum threshold”).
                                 Warning().
                                 Name(“LaserVIEW Max Threshold”).
                                 Time(_max.time));

            _max = (0, int.MinValue);
            return ProcessReturns.
                        SuccessAndSendData($”Processed layer {endOfLayerData.Layer.Value}”,
                                           alerts);
        …
      }
…
```

NOTE: Alert descriptions returned to DataHUB are forwarded to Renishaw Central, which may trigger actions such as sending an SMS.

6.4.10.4 Final implementation

The complete plug-in is as follows:

```
public class PlugIn
{
   public static string APIVersion() => "2";
   public static string DisplayName() => "Tutorial Example 5";
   public string Initialise(initialisationData)
   {
      var iv               = JObject.Parse(initialisationData);
      var id               = JObject.Parse(iv["InitialisationData"].Value<string>());
      var example5InitData = id["PlugIn.Tutorials.v2.Example5"].ToObject<InitialisationData>();
      _alertMaxThreshold = example5InitData.AlertMaxThreshold;
      return InitialiseReturns.Success();
   }
   public string Process(IReadOnlyCollection<Token> tokens)
   {
      foreach (var token in tokens)
         switch (token)
         {
            case Sample sample:
               if (sample.LaserVIEW >= _max.value)
                  _max = (sample.Time, sample.LaserVIEW);
               break;
            case EndOfLayerLaserData endOfLayerLaserData:
               return ProcessReturns.Success($"Processed layer {endOfLayerLaserData.Layer.Value}, "
                                             + $"laser {endOfLayerLaserData.Laser.Value}");
            case EndOfLayerData endOfLayerData:
               var alerts = new List<Alert>();
               if (_max.value > _alertMaxThreshold)
                  alerts.Add(Alert.Factory().
                                    Description("LaserVIEW intensity has exceeded the maximum threshold").
                                    Warning().
                                    Name("LaserVIEW Max Threshold").
                                    Time(_max.time));
               _max = (0, int.MinValue);
               return ProcessReturns.
                       SuccessAndSendData($"Processed layer {endOfLayerData.Layer.Value}", alerts);
         }
      return ProcessReturns.SuccessWithoutInformation();
```

```
    }
    public string Shutdown() => ShutdownReturns.Success();
    private (uint time, int value) _max = (0, int.MinValue);
    private int _alertMaxThreshold;
}
```

6.4.11 The Plug-in API V2.0

For an assembly to be usable by the DataHUB service as a V2.0 plug-in, the following public API must be implemented.

6.4.11.1 Assembly naming

The .NET assembly file name must start with “PlugIn.” (“PlugIn” followed by a full-stop) and have the extension “dll”.

6.4.11.2 Plug-in class naming

The plug-in assembly must define a public class named “PlugIn”.

6.4.11.3 Plug-in class methods

The plug-in class must implement the following public methods:

```
public PlugIn()
public static string APIVersion()
public static string DisplayName()
public string Initialise(string data)
public string Process(IReadOnlyCollection<Token> data)
public string Shutdown()
```

6.4.11.4 The parameterless constructor

The type must have a parameterless constructor. If no constructor(s) with parameters are defined, C# provides this automatically, i.e., there is no need to define it explicitly.

NOTE: A plug-in should not acquire any resources inside its constructor.

A plug-in instance that is constructed is not guaranteed to have its *Shutdown* called, therefore any such resources may not be cleaned up in a timely manner. *Initialise* is the appropriate place to acquire resources.

6.4.11.5 The static `APIVersion` method

The static `APIVersion` method is used by DataHUB to identify which version of the API this plug-in should be validated against, and hence the format of the process monitoring data it will process.

Parameters:

None.

Return: `string`

A plug-in that implements API V2.0 must return the literal “2”.

6.4.11.6 The static `DisplayName` method

The content of the string returned by the static `DisplayName` method is used as the plug-in name displayed in DataHUB Monitor, in creating the plug-in's output folder, when auditing events for the plug-in, and so on. Although it is not mandatory for this name to be unique, making it so helps identify the plug-in when, for example, multiple versions are installed on a DCPC.

Parameters:

None.

Return: `string`

The literal to use in naming the plug-in in various DataHUB UX and logs.

6.4.11.7 The `Initialise` method

This method is called by DataHUB at the start of a build, before any layer data is sent to the plug-in. Hence it is passed *build invariant* data to the plug-in. The plug-in may use this method to allocate resources required during the lifetime of the instance, calculate build constants, and so on.

Parameter: `string`

A string containing a JSON object conforming to the following schema:

```
{
   "AssemblyFolder":     string
   "OutputFolder":       string
   "NumberOfLasers":     unsigned integer
   "InitialisationData": JSON object
}
```

`AssemblyFolder:` `string`

The path to the containing folder of the assembly that contains this plug-in:

"C:\ProgramData\Renishaw\DataHUB\PlugIns\PlugIn.Example"

`OutputFolder:` `string`

The path to the folder to which the plug-in instance should write any output files. Each plug-in instance run for a job will have a unique folder generated for them within the job's output folder. The name of this folder will be constructed from the plug-ins display name and API version and a time stamp and GUID, using the following structure:

"\[2] <Display name>.<Time stamp in the format yyyMMdd-hhmmss.fff> (<GUID>)"

`NumberOfLasers:` `unsigned integer`

This will be a value from 1 to 4, as defined by the hardware configuration of the machine creating the data.

InitialisationData: JSON object

The content provided during the “Plug-in Initialisation data” stage of the job creation workflow in DataHUB Generator. See “The initialisation data string” on page 6-97, for more detail on the structure and usage of this string.

Return: string

A string containing a JSON object conforming to the following schema:

```
{
   “result”: string
   “message”: string
   “timeSeries”:
   [
      {
         “name”:               string
         “unitType”:           string
         “unit”:               string
         “absoluteOrRelative”: string
         “component”:          string
         “displayPrecision”:   unsigned integer
         “graphType”:          string
         “grouping”:           string
         “precision”:          unsigned integer
         [Any number of additional custom properties]
       }
    ]
 }
```

result: string

Whether the plug-in successfully initialised or not. It must be either:

- the literal value “Success” which indicates that the plug-in has successfully initialised and can begin processing, or
- the literal value “Failure” which indicates that the plug-in has encountered an error and will be shut down without receiving data.

This property is mandatory.

message: string

A message containing more detail that will be appended to the result and recorded in the audit log. This property is optional – if not present only the result will be recorded in the audit log.

timeSeries: JSON array

An array containing the definitions of time series to which the plug-in will publish data to during processing. This property is optional – if not present or an empty array, DataHUB will not create any time series in Renishaw Central.

timeSeries[n].name: string

The name of the time series. This property is mandatory.

timeSeries[n].unitType: string

The family of units the data represents. This allows downstream conversion to a preferred unit type for display. Pre-defined unit types are:

- Acceleration
- Angle
- Binary
- Distance
- Flowrate
- Humidity
- MicroDistance
- Percentage
- Pressure
- Speed
- Temperature
- Dimensionless

This property is mandatory.

timeSeries[n].unit: string

By convention, Renishaw Central prefers SI units, expects “per” relationships to be indicated using the ‘/’ character, and formats powers as number. For example, acceleration would usually be represented as m/s^2. For temperature and degrees, use the ‘°’ character where relevant, for example °C for Celsius. Where the values do not have a dimension, use the literal value `Dimensionless`. This property is mandatory.

timeSeries[n].absoluteOrRelative: string

A hint to applications as to whether values are absolute or relative to the previous value. It must be either:

- the literal value “Absolute” which indicates that the values are absolute, or
- the literal value “Relative” which indicates that the values are relative to the previous value.

This property is optional – if it is not present it will be given the value Absolute.

timeSeries[n].component: string

Associates the time series to a distinct aspect of a device, commonly a physical entity such as a weather station, tool or software system. This property is optional – if it is not present it will be given the value “Machine”.

`timeSeries[n].description: string`

A description of the time series. This property is optional.

`timeSeries[n].displayHintPrecision: unsigned integer`

A hint to applications that integrate with Renishaw Central as to what precision to display values in the time series at. This property is optional.

`timeSeries[n].displayHintSampleRate: string`

A hint to applications that integrate with Renishaw Central as to what sampling rate to use. If omitted, most downstream apps will assume to use ‘default’. The well-known sample rates are:

- Raw
- Default
- Hour
- Day
- Week
- Month

This property is optional.

`timeSeries[n].graphType: string`

A hint to applications that integrate with Renishaw Central as to type of graph the time series should be plotted as. The well-known types of graphs are:

- Bar
- Binary (specifically to display On-Off data)
- Line

This property is optional.

`timeSeries[n].grouping: string`

The group this time series should be collected with. This property is optional.

`timeSeries[n].precision: unsigned integer`

A record of the precision the data was captured/calculated at. This property is optional.

`timeSeries[n]` *Additional Properties*

The plug-in may include any number of additional properties that will be stored in Renishaw Central and can be accessed by custom software. They cannot use the reserved names “machine” or “source”.

If the returned string does not conform to the above schema, this is treated as an error and the full string is recorded to the audit log.

6.4.11.8 The Process method

DataHUB passes subsections of the build data in sequential calls: the total number of calls will vary dependent on size of the data. In this method the plug-in should perform the bulk of its analysis work.

Parameter: IReadOnlyCollection<Token>

A section of the build token stream.

Return: `string`

An empty string, or a string containing either “Success” or “Failure”, or a string containing a JSON object conforming to the following schema:

```
{
   “result”:  string
   “message”: string
   “alerts”:
   [
      {
         “description”: string
         “level”:    string
         “name”:    string
         “time”:    unsigned integer
      }
   ],
   “analysisResults”:
   [
      {
         “name”: string
         “time”: unsigned integer
         Any number of additional custom properties
      }
   ],
   “timeSeries”:
   [
      {
         “name”: string
         “unitType”: string
         “unit”: string
         “values”:
         [
            {
                “time”:  unsigned integer
```

```
                "value": number
            }
        ],
        "limits":
         [
            {
               "time":        unsigned integer
               "lowerDisplayLimit":  number
               "upperDisplayLimit":  number
               "lowerWarningLimit":  number
               "upperWarningLimit":  number
               "lowerOperatingLimit": number
               "upperOperatingLimit": number
            }
        ]
     }
   ]
}
```

result: string

Whether the plug-in successfully processed the data or not. It must be either

- the literal value Success, which indicates that the plug-in has successfully processed, or
- the literal value Failure, which indicates that the plug-in has encountered an error. The plug-in will remain active and continue to receive subsequent data.

This property is mandatory.

message: string

A message containing more detail that will be appended to the result and recorded in the audit log. This property is optional – if not present nothing will be recorded in the audit log.

alerts: JSON Array

An array containing the alerts the plug-in wishes to publish to Renishaw Central during processing. This property is optional – if not present or an empty array, DataHUB will not attempt to publish any alerts to Renishaw Central.

alerts[n].description: string

A description providing details about the alert. This property is mandatory.

alerts[n].level: string

One of the following literal values indicating the severity of the alert:

- Info
- Warning
- Error

This property is mandatory.

alerts[n].name: string

The name of the alert. This property is mandatory.

alerts[n].time: unsigned integer

The time the alert occurred, in microseconds relative to the start time of the layer. This property is mandatory.

analysisResults: JSON Array

An array containing the analysis results the plug-in wishes to publish to Renishaw Central during processing. This property is optional – if not present or an empty array, DataHUB will not attempt to publish any analysis results to Renishaw Central.

analysisResults [n].name: string

The name of the analysis result. This property is mandatory.

analysisResults [n].time: unsigned integer

The time the analysis result occurred, in microseconds relative to the start time of the layer. This property is mandatory.

analysisResults[n] *Additional Properties*

The analysis result may include any number of additional properties, that will be stored in Renishaw Central and can be accessed by custom software. They may not, however, use any of the following reserved names:

- created
- machine
- source
- type

timeSeries: JSON Array

An array containing time-series data the plug-in wishes to add to the time series defined in the *Initialise* method. This property is optional – if not present or an empty array, DataHUB will not update the data of any time series in Renishaw Central.

timeSeries[n].name: string

The name of the time series as it was defined at initialisation. This property is mandatory.

timeSeries[n].unitType: string

The type of the time series as it was defined at initialisation. This property is mandatory.

timeSeries[n].unit: string

The unit of the time series as it was defined at initialisation. This property is mandatory.

timeSeries[n].values: JSON Array

An array containing the time-value pairs to add to the time series. A time series must have either a values property, a limits property, or both.

timeSeries[n].values[m].time: unsigned integer

The time the value corresponds to, in µs relative to the start time of the layer. This property is mandatory.

timeSeries[n].values[m].value: number

The value. This property is mandatory.

timeSeries[n].limits: JSON Array

An array containing the limits of data. There are three sets of limits:

1. Operating
2. Warning
3. Display

Applications that integrate with Renishaw Central may use these limits to aid display and to trigger actions when time-series data points exceed these various limits.

Limits take effect from the time given by the time property until the time given by the next limits added – the last defined limits take effect in perpetuity.

It is not necessary that a plug-in provide limits for a time series – it may be that the data has no natural operation range or sensible display range for example. It is also not necessary that the limits ever change once set – the opportunity is provided for cases in which a change would be appropriate.

A time series must have either a values property, a limits property, or both.

timeSeries[n].limits[m].time: `unsigned integer`

The time the limit should take effect from, in µs relative to the start time of the layer. This property is mandatory.

timeSeries[n].limits[m].lowerDisplayLimit: `number`

The value of the lower display limit. This property is optional – if it is not present, the lower display limit will not be changed from its current value.

timeSeries[n].limits[m].upperDisplayLimit: `number`

The value of the upper display limit. This property is optional – if it is not present, the upper display limit will not be changed from its current value.

timeSeries[n].limits[m].lowerWarningLimit: `number`

The value of the lower warning limit. This property is optional – if it is not present, the lower warning limit will not be changed from its current value.

timeSeries[n].limits[m].upperWarningLimit: `number`

The value of the upper warning limit. This property is optional – if it is not present, the upper warning limit will not be changed from its current value.

timeSeries[n].limits[m].lowerOperationalLimit: `number`

The value of the lower operation limit. This property is optional – if it is not present, the lower operational limit will not be changed from its current value.

timeSeries[n].limits[m].upperOperationalLimit: `number`

The value of the upper operation limit. This property is optional – if it is not present, the upper operational limit will not be changed from its current value.

If the returned string does not conform to the above schema, this is treated as an error and the full string is recorded to the audit log.

6.4.11.9 The *Shutdown* method

The *Shutdown* method is called exactly once and will be called either when the *Initialise* method returns an error, or when there is no further data to process (because all expected data has been processed, or the job was cancelled). Once called, the plug-in will receive no further calls of any kind.

In this method, the plug-in should perform any final processing on the build data, release any resources it may have acquired, and so on.

Parameters: None

`Return: string`

If the string contains the word “Success” it indicates that the plug-in has successfully shutdown. If it does not, it indicates that the plug-in has encountered an error. In either case, the string is recorded in full to the audit logs.

6.4.11.10 The initialisation data string

DataHUB initialises a V2.0 plug-in with a JSON-formatted string containing values for global build parameters, and the plug-in initialisation string specified in the DataHUB Generator workflow for the plug-in processing job for the build.

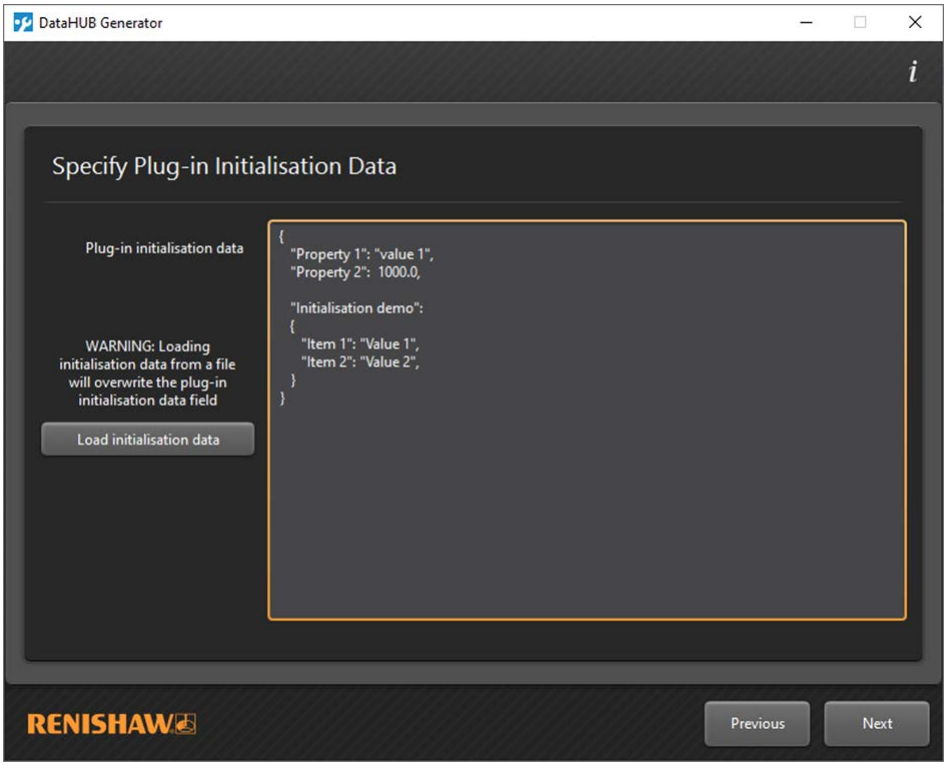


Figure 42 Plug-in input JSON string

The string entered in the DataHUB Generator workflow is itself JSON formatted. By adding properties and objects, data specific to each plug-in type can be sent to the instance’s *Initialise* method.

An example plug-in initialisation string might be as follows:

```
{
   “Property 1”: “value 1”,
   “Property 2”:  1000.0,
   “Initialisation demo”:
   {
      “Item 1”: “Value 1”,
      “Item 2”: “Value 2”,
   }
}
```

Using a JSON parsing library (e.g., Newtonsoft.Json or System.Text.Json) in the plug-in simplifies the process of extracting the values contained in the initialisation string, for example:

```
public string Initialise(string initialisationData)
{
   var instanceData         = JObject.Parse(initialisationData);
   var assemblyFolder       = instanceData[“AssemblyFolder”];
   var outputFolder         = instanceData[“OutputFolder”];
   var numberOfLasers       = instanceData[“NumberOfLasers”];
   var dataHUBGeneratorData = JObject.Parse(instanceData[“InitialisationData”].Value<string>());
   var property1            = dataHUBGeneratorData[“Property 1”];
   var property2            = dataHUBGeneratorData[“Property 2”];
   var item1                = dataHUBGeneratorData[“Initialisation demo”][“Item 1”];
   var item2                = dataHUBGeneratorData[“Initialisation demo”][“Item 2”];
```

The instance initialisation data specified in the DataHUB Generator is itself a JSON string, so note that it is parsed as a JObject before accessing the properties contained. The values of properties `Property1` and `Property2`, and the property values in the nested `Initialisation` demo object are shown in the VisualStudio watch window:

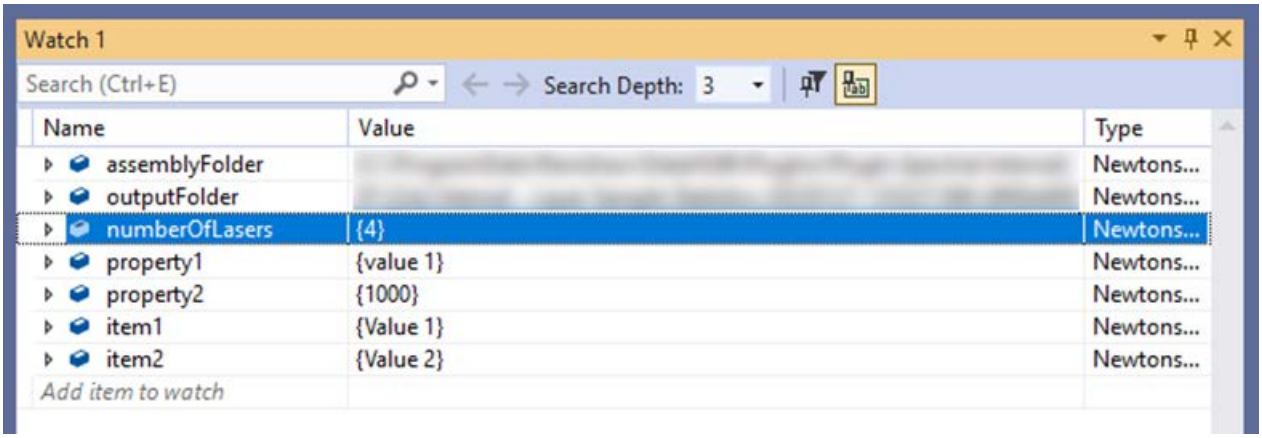


Figure 43 Plug-in initialisation data string in Initialise call

Multiple plug-ins are usually in use on a DCPC, so a recommended strategy for formatting the DataHUB Generator string is to specify a JSON object named for each plug-in type:

```
{
   “PlugIn.TypeA”:
   {
      “PlugIn parameter 1”:1,
      “PlugIn parameter 2”:2,
   },
   “PlugIn.TypeB”:
  {
      “PlugIn parameter 1”:100,
      “PlugIn parameter 2”:200,
   }
}
```

Accessing the initialisation data specific to a plug-in is then a matter of accessing the relevant nested object, i.e., in plug-in TypeA and TypeB respectively:

```
var dataA = dataHUBGeneratorData[“PlugIn.TypeA”];
var dataB = dataHUBGeneratorData[“PlugIn.TypeB”];
```

6.4.11.11 Token types

6.4.11.11.1 DataSourceInformation

This token describes features of a process monitoring data source.

```
public string      File { get; }
public LayerNumber Layer { get; }
public LaserID     Laser { get; }
public DateTime    LayerStartTime { get; }
public double      MillisecondsPerSample { get; }
public uint        TotalNumberOfSamples { get; }
```

6.4.11.11.2 EndOfLayerData

This token denotes the end of all data for a layer.

```
public LayerNumber Layer { get; }
public uint        LastSampleTime { get; }
```

6.4.11.11.3 EndOfLayerLaserData

This token marks the end of the samples for a laser, for a layer.

```
public LayerNumber Layer { get; }
public LaserID     Laser { get; }
```

6.4.11.11.4 PacketOfSamples

This token contains a collection of samples.

```
public IreadOnlyList<Sample> Samples { get; }
```

6.4.11.11.5 Sample

A sample token contains data collected at a discrete point in time during the layer build process. For a layer, for a laser, there will be many sample tokens.

- `public uint   Time               { get; }`
- `public double DemandX            { get; }`
- `public double DemandY            { get; }`
- `public double DemandFocus        { get; }`
- `public int    DemandLaserPower   { get; }`
- `public int    LaserModulate      { get; }`
- `public int    MeltVIEWPlasma     { get; }`
- `public int    MeltVIEWMeltpool   { get; }`
- `public int    LaserVIEW          { get; }`
- `public int    LaserOutputPower   { get; }`
- `public int    LaserBackReflection { get; }`

6.4.11.11.6 Split

This token has no properties.

6.4.11.11.7 StartOfLayerData

This token marks the start of the series of tokens pertaining to a layer.

```
public LayerNumber Layer { get; }
```

6.4.11.11.8 StartOfLayerLaserData

This token marks the start of the series of tokens for a laser for a layer.

```
public LayerNumber Layer { get; }
public LaserID     Laser { get; }
```

6.4.11.11.9 SummaryPacket

This token contains summary values produced by summary methods provided to the `SummarisePacketOfSamples` filter. The set of summary values are contained in instances of the nested class `SummaryValues`. As summary methods may produce fractional value results, the type of each property in `SummaryValues` is double.

```
Public uint Time   { get; }
public uint Duration { get; }
public IreadOnlyDictionary<string, SummaryValues> Summary { get; }
public class SummaryValues
{
   public double DemandX             { get; }
   public double DemandY             { get; }
   public double DemandFocus         { get; }
   public double DemandLaserPower    { get; }
   public double MeltVIEWPlasma      { get; }
   public double MeltVIEWMeltpool    { get; }
   public double LaserVIEW           { get; }
   public double LaserOutputPower    { get; }
   public double LaserBackReflectio  { get; }
}
```

6.4.11.11.10 Token

The base class from which all tokens derive. This class contains no data.

6.4.11.11.11 LaserID and LayerNumber

The underlying type of laser IDs and layer numbers is integer, so to prevent accidental misassignment of, for example, a layer number to a laser ID, these provide type safety when constructing the various tokens above.

6.4.11.12 Filters

The API provides a set of filters designed to perform common tasks when processing a token stream.

6.4.11.12.1 EmitSampleOnlyIf

This filter compares the value of one of an input `Sample` token's property and will emit the `Sample` only if the value passes the test. All token types other than `Sample` are emitted to the output stream.

The following property value are offered:

- DemandX
- DemandY
- DemandFocus
- DemandLaserPower
- MeltVIEWPlasma
- MeltVIEWMeltpool
- LaserVIEW
- LaserOutputPower
- LaserBackReflection
- LaserModulate
- The equality tests are:
- EqualTo
- NotEqualTo
- GreaterThan
- GreaterThanOrEqualTo
- LessThan
- LessThanOrEqualTo
- CustomTest
- IsOn (LaserModulate only)

The final variant of the filter allows a customisable comparison test to be configured. Any function that takes two integer inputs and returns a Boolean can be used, for example:

```
var customTest = EmitSampleIf.DemandX.CustomTest(v => v % 2 == 0);
```

The custom test here would pass samples with a DemandX value that is even.

6.4.11.12.2 SplitWhen

This filter compares the value of one of an input `Sample` token's property value to the property's value in the preceding token, and if different will add a `Split` token before outputting the input token, effectively adding a `Split` between the two tokens. All token types other than `Sample` are emitted to the output stream.

The following variants are available:

- DemandXChanges
- DemandYChanges
- DemandFocusChanges
- LaserModulateChanges
- DemandLaserPowerChanges

NOTE: Only demand properties are provided; the other process monitoring properties have a high variability, sample to sample, and therefore are deemed unsuitable for splitting the token stream.

6.4.11.12.3 CollateInto

This filter gathers all `Sample` tokens bounded by `Split` tokens into a single `PacketOfSamples` token. `Split` tokens and `Sample` tokens are swallowed. All other token types are emitted to the output stream.

```
public static Func<Token, IEnumerable<Token>> PacketOfSamples()
```

The scope of this mapping is worth noting. Given a stream of tokens previously divided by `Split` tokens (for example, when laser modulate, positions, etc. changed), the resulting output stream contains no `Sample` or `Split` tokens, these having been replaced by `PacketOfSamples` tokens.

6.4.11.12.4 SummarisePacketOfSamples

This filter processes the `PacketOfSamples` tokens produced by `CollateInto` and performs one or more summary functions on the collated sample data values. The constructor of this filter allows one or more named summary functions to be configured, and the results of each are stored in the output token.

```
public static Func<Token, IEnumerable<Token>> Summarise(params SummaryMethod[] summaryMethods)
```

A SummaryMethod instance is constructed with a name and a summary function:

```
public SummaryMethod(string description, Func<IReadOnlyList<int>, double> func)
```

6.4.11.12.5 Pipeline

To build filters into pipelines, these two helpers are necessary:

```
public static Func<Token, IEnumerable<Token>> First(<Token, IEnumerable<Token>> f1);
public static Func<Token, Ienumerable<Token>> Then(Func<Token, IEnumerable<Token>> f0,
                                                   Func<Token, IEnumerable<Token>> f1);
```

6.4.11.13 Helpers

The API provides some classes defining common operations used when writing plug-ins.

6.4.11.13.1 PositionHelpers

This class provides functions to round double values in a consistent manner:

```
public static double RoundToMicrons(double value, double microns);
public static int Round(double value);
```

6.4.11.13.2 Return string helpers

The API provides helper methods to build up the return strings required for DataHUB at each stage of the plug-in (*Initialise*, *Process* and *Shutdown*).

NOTE: DataHUB expects to receive a valid JSON return string that conforms to the return string API. It is important that special characters are escaped with an extra backslash in strings. Not doing so can result in an invalid JSON string being sent to and discarded by DataHUB. For example:

```
ProcessReturns.Success("this is a badly \"escaped\" message");
ProcessReturns.Success("this is a correctly \\\"escaped\\\" message");
```

A second example:

```
ProcessReturns.Success("this is a badly \nescaped message");
ProcessReturns.Success("this is a correctly \\nescaped message");
```

6.4.11.13.3 Initialise Returns

Helpers to build simple plug-in return strings from Initialise back to DataHUB are as follows:

```
public static string Success();
public static string Success(string message);
public static string Success(IReadOnlyCollection<string> messages);
public static string Failure();
public static string Failure(string message);
public static string FailureDueToThisException(Exception e);
```

To create one or more time series in Renishaw Central, the following helper will can be used:

```
public static string SuccessAndCreateTimeSeries(IEnumerable<TimeSeries> timeSeries)
public static string SuccessAndCreateTimeSeries(string message, IEnumerable<TimeSeries> timeSeries)
```

A `TimeSeries` object must be constructed with a name, unit type, and a unit. The `TimeSeries` object factory provides a fluent interface with well-known unit types and units to make construction a simple, less error prone process. This is demonstrated in the example below:

```
var titaniumTimeSeries = TimeSeries.Factory().
                                    Name("Thermal expansion").
                                    Temperature().
                                    Celsius().
                                    Grouping("Titanium").
                                    Create();
```

6.4.11.13.4 Process Returns

Helpers to build simple plug-in return strings from *Process* back to DataHUB are as follows:

```
public static string Success();
public static string SuccessWithoutInformation();
public static string Success(string message);
public static string Success(IReadOnlyCollection<string> messages);
public static string Failure();
public static string Failure(string message);
public static string FailureDueToThisException(Exception e);
```

With the following helpers, plug-in return strings from the Process function can be used to signify success or failure and send data such as time-series updates, alerts, and analysis results to Renishaw Central.

```
public static string SuccessAndSendData(string message, IEnumerable<UpdateTimeSeries> timeSeries)
public static string SuccessAndSendData(string message, IEnumerable<Alert> alerts)
public static string SuccessAndSendData(string message,
                                        IEnumerable<AnalysisResult>
                                        analysisResults)
public static string SuccessAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string SuccessAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResult)
public static string SuccessAndSendData(string message,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
```

```
                                        IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries)
public static string SuccessAndSendData(IEnumerable<Alert> alerts)
public static string SuccessAndSendData(IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResult)
public static string SuccessAndSendData(IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries)
public static string FailureAndSendData(IEnumerable<Alert> alerts)
public static string FailureAndSendData(IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message, IEnumerable<UpdateTimeSeries> timeSeries)
public static string FailureAndSendData(string message, IEnumerable<Alert> alerts)
public static string FailureAndSendData(string message, IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string FailureAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
```

```
                                        IEnumerable<AnalysisResult> analysisResults)
```

TimeSeries that were defined during *Initialise* can be updated with values, limits, or both. The *TimeSeries* object provides an Update method that returns an UpdateTimeSeries object as demonstrated in the example below:

```
var update = titaniumTimeSeries.
      Update().
      WithValues((0, 32), (100, 67)).
      WithLimits(TimeSeriesLimit.Factory().
                                Time(0).
                                LowerDisplayLimit(0).
                                NoUpperDisplayLimit().
                                LowerWarningLimit(30).
                                NoUpperWarningLimit().
                                LowerOperatingLimit(10).
                                NoUpperOperatingLimit());
```

Alerts can be raised on Renishaw Central by constructing an Alert object and passing it to the appropriate SendData helper method. The Alert factory provides a simple builder interface as shown in the example below:

```
var overheatingAlert = Alert.Factory().
                            Description("Part temperature has exceeded the threshold").
                            Warning().
                            Name("Overheating").
                            Time(350);
```

Finally, analysis results can also be raised on Renishaw Central by constructing an AnalysisResult object and passing it to the appropriate Send Data helper method. The AnalysisResult factory provides a simple builder interface as shown in the example below:

```
var analysisResult = AnalysisResult.Factory().
                                    Name("Thermal expansion rate").
                                    Time(250).
                                    AddAdditionalProperty("Rate", 0.05).
                                    Create();
```

6.4.11.13.5 Shutdown Returns

Helpers to build simple plug-in return strings from *Shutdown* back to DataHUB are as follows:

```
public static string Success();
public static string Success(string message);
public static string Success(IReadOnlyCollection<string> messages);
public static string Failure();
public static string Failure(string message);
public static string FailureDueToThisException(Exception e);
```

6.5 Export packets plug-in

The LaserVIEW and MeltVIEW hardware collect data sampled at a rate of 100 kHz, with each discrete sample containing several values. While data at this resolution is useful in some analysis tasks, a lower resolution representation of the same samples may be adequate in many analysis tasks, so the plug-in quantises the data into “packets”, where a packet is defined by:

- the laser is firing, and
- the demand X and Y positions are constant.

This definition effectively gathers the set of samples taken during a single exposure of the laser on the powder bed. From each set, both mean and median summary values of the Laser/MeltVIEW process monitoring sensor values are produced. It should be noted that, due to the operational characteristics of the laser, samples at the start of an exposure may contain slightly low Laser/MeltVIEW values. The median value may therefore be a better summary representative for the sample data than the mean value.

6.5.1 Running the plug-in

To use this plug-in, run DataHUB Generator and select the *Build Data* folder containing the build process monitoring files. Select a folder for the plug-in output. If necessary (for example, the build information file *BuildStarted yyyy.mm.dd.HH.mm.ss.dhxml* was not found in the input folder), ensure the correct *Number of Lasers* is set as appropriate for the build in question – the plug-in will not process data if there is a mismatch between the selected number of lasers and the actual number of lasers equipped with LaserVIEW/MeltVIEW hardware on the AM machine that produced the data. The two other values, *Layer Spacing* and *Build Height* can be left untouched as they are not required by the plug-in.

Enter a description and press Next.

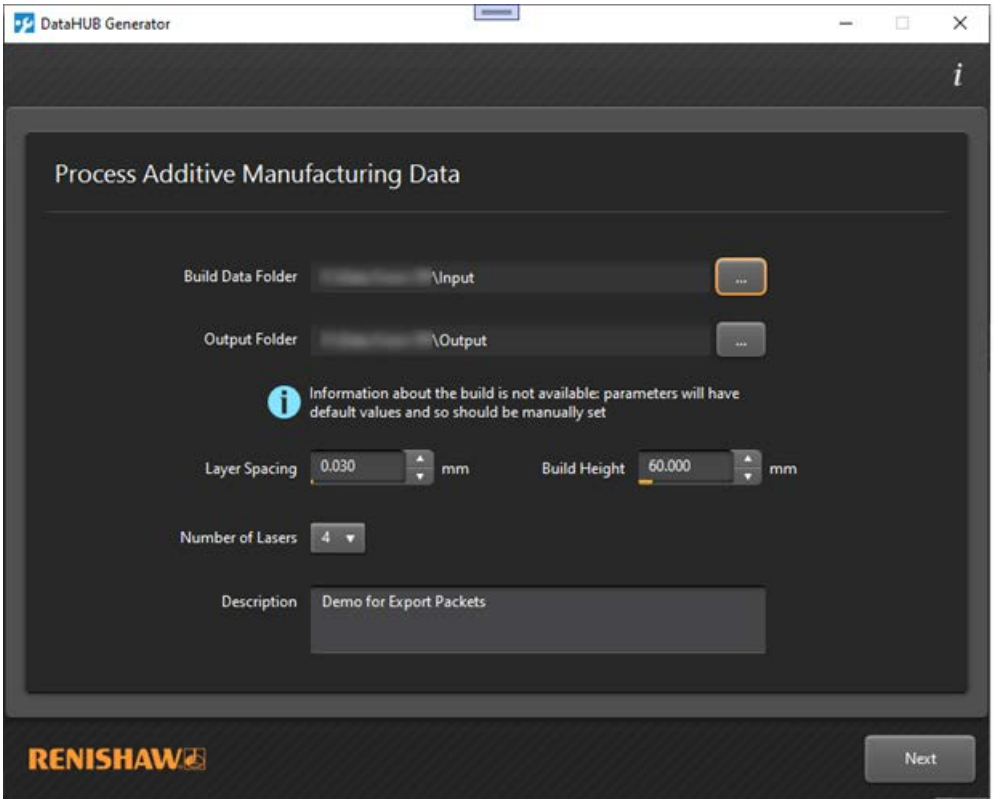


Figure 44 Workflow stage 1 – configure data

In the next part of the workflow, select “Process LaserVIEW and/or MeltVIEW data via plug-ins”, and press Next.

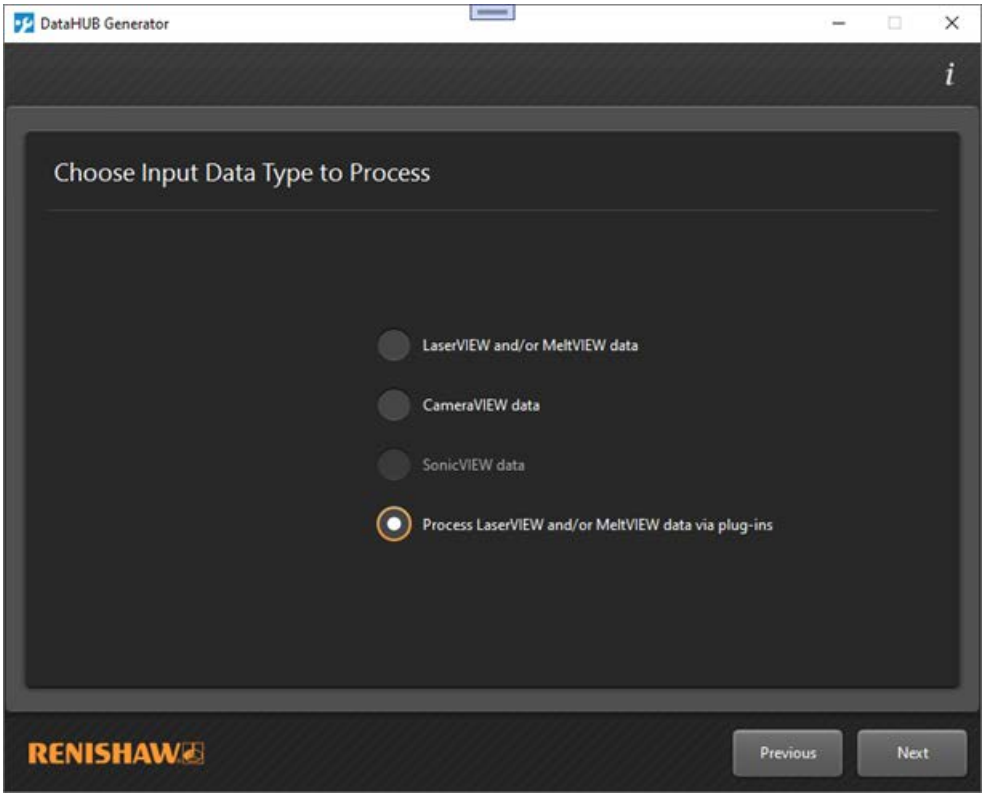


Figure 45 Workflow stage 2 – select plug-in processing

As this plug-in requires no initialisation data, the text box can be left empty. Press Next.

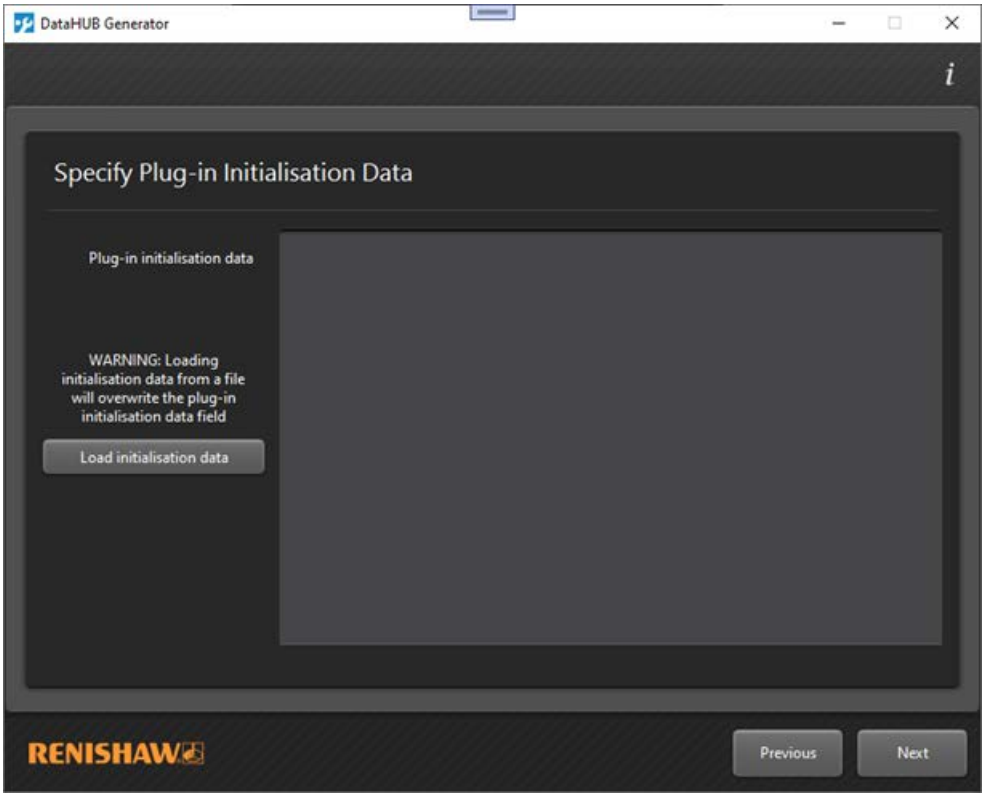


Figure 46 Workflow stage 3 – plug-in initialisation

In the final part of the workflow, press “Trigger LaserVIEW/MeltVIEW Plug-in Processing”.

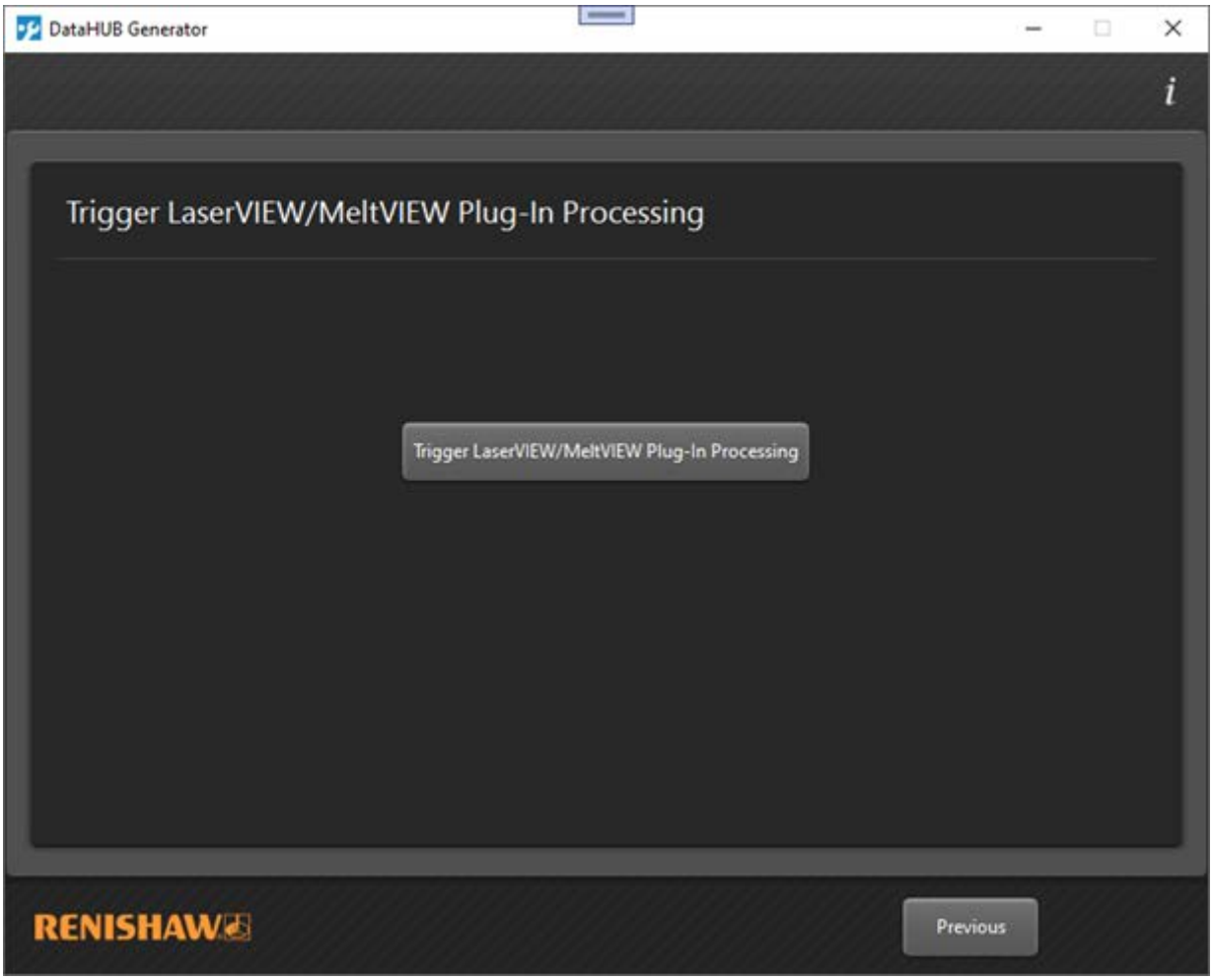


Figure 47 Workflow stage 4 - Start processing

The DataHUB Generator app can now be exited.

6.5.2 File name and location

The *Output* folder specified in the workflow forms the root folder used by the plug-in to write its output. The plug-in creates a subfolder named in the following fashion:

[2] Export Packets yyyyMMdd-HHmmss.ff (GUID)

Where “yyyyMMdd-HHmmss.ff” is the date and time at which the plug-in started processing the data, and “GUID” is a unique code. This naming has been chosen to ensure re-running the plug-in over the same data does not overwrite any pre-existing plug-in output.

In the subfolder, the plug-in will write a file per laser, per layer, named “Packet data for layer x, laser y.txt”, where x is the layer number and y the laser number. Note that although the file uses the “txt” extension, it is in simple tab-delimited form so may be imported into many software packages with no pre-processing.

6.5.3 File contents

Each file begins with a row of column headings:

- Start time: relative to the start of the layer (µs)
- Duration: derived from the times of the first and last samples gathered in the packet (µs)
- Demand X: the packet position on the powder bed (left –125.0 to right +125.0 mm)
- Demand Y: the packet position on the powder bed (front –125.0 to back +125.0 mm)
- Demand focus: the focusing of the laser beam on the powder bed (±5.0 mm)
- Demand laser power (mean): the averaged values of the samples gathered in the packet
- MeltVIEW plasma (mean): the averaged values of the samples gathered in the packet
- MeltVIEW melt pool (mean): the averaged values of the samples gathered in the packet
- LaserVIEW (mean): the averaged values of the samples gathered in the packet
- Laser back reflection (mean): the averaged values of the samples gathered in the packet
- Laser output power (mean): the averaged values of the samples gathered in the packet
- Demand laser power (median): the median value of the samples gathered in the packet
- MeltVIEW plasma (median): the median value of the samples gathered in the packet
- MeltVIEW melt pool (median): the median value of the samples gathered in the packet
- LaserVIEW (median): the median value of the samples gathered in the packet
- Laser back reflection (median): the median value of the samples gathered in the packet
- Laser output power (median): the median value of the samples gathered in the packet

There then follows a series of rows containing the summarised values for each packet of AMPM data:

Start time	Duration	Demand X	Demand Y	Demand focus	Demand laser power (mean)	MeltVIEW plasma (mean)	MeltVIEW melt pool (mean)	LaserVIEW (mean)	Laser back reflection (mean)	Laser output power (mean)	Demand laser power (median)	MeltVIEW plasma (median)	MeltVIEW melt pool (median)	LaserVIEW (median)	Laser back reflection (median)	Laser output power (median)
29800	20	-110.365	-76.35	0	2418	59.5	59	167	12801	16216	2418	59.5	59	167	12801	16216
29830	20	-110.36	-76.325	0	2418	631.5	371	696.5	17184	19679.5	2418	631.5	371	696.5	17184	19679.5
29860	20	-110.365	-76.29	0	2418	1017	672	725.5	17709.5	20158.5	2418	1017	672	725.5	17709.5	20158.5
29890	20	-110.365	-76.265	0	2418	902	653.5	729.5	17847.5	20287.5	2418	902	653.5	729.5	17847.5	20287.5
29920	20	-110.37	-76.235	0	2418	962.5	746.5	735	9562	7149	2418	962.5	746.5	735	9562	7149
30970	20	-110.26	-76.14	0	2418	379.5	434	424	15568.5	18501	2418	379.5	434	424	15568.5	18501
31000	20	-110.265	-76.165	0	2418	717.5	432.5	725.5	17634.5	20103	2418	717.5	432.5	725.5	17634.5	20103
31030	20	-110.265	-76.19	0	2418	1022	684	730	17855	20304.5	2418	1022	684	730	17855	20304.5
31060	20	-110.27	-76.22	0	2418	964	725.5	737.5	17891.5	20339.5	2418	964	725.5	737.5	17891.5	20339.5
31090	20	-110.27	-76.245	0	2418	975	756.5	735	17934.5	20400.5	2418	975	756.5	735	17934.5	20400.5
31120	20	-110.27	-76.275	0	2418	1180.5	913.5	739	17960.5	20414	2418	1180.5	913.5	739	17960.5	20414
31150	20	-110.265	-76.3	0	2418	1196	922.5	736	17984	20421	2418	1196	922.5	736	17984	20421
31180	20	-110.265	-76.33	0	2418	1163	888	740.5	17965.5	20420	2418	1163	888	740.5	17965.5	20420
31210	20	-110.265	-76.36	0	2418	1077.5	853	734.5	17973	20416	2418	1077.5	853	734.5	17973	20416
31240	20	-110.26	-76.385	0	2418	1098.5	875.5	743.5	17963.5	20403.5	2418	1098.5	875.5	743.5	17963.5	20403.5
31270	20	-110.265	-76.415	0	2418	1165.5	947	737	17928	20363	2418	1165.5	947	737	17928	20363
31300	20	-110.265	-76.445	0	2418	1086	873.5	741.5	9582	7164	2418	1086	873.5	741.5	9582	7164
32350	20	-110.175	-76.535	0	2418	418	456	428	15556.5	18509	2418	418	456	428	15556.5	18509
32380	20	-110.175	-76.505	0	2418	834	490	728	17640	20111	2418	834	490	728	17640	20111
32410	20	-110.175	-76.475	0	2418	1060	711	742	17858	20319	2418	1060	711	742	17858	20319
32440	20	-110.175	-76.45	0	2418	1125	812	741.5	17918	20383	2418	1125	812	741.5	17918	20383
32470	20	-110.165	-76.42	0	2418	1210	894.5	744.5	17959.5	20409.5	2418	1210	894.5	744.5	17959.5	20409.5
32500	20	-110.17	-76.39	0	2418	1202	860	745	17952	20387	2418	1202	860	745	17952	20387

This page is intentionally left blank.

RENISHAW
apply innovation™

www.renishaw.com/additivemanufacturing

#renishaw

+44 (0) 1453 524524 uk@renishaw.com

Part no.: H-5800-4762-01-A
Issued: 06.2023