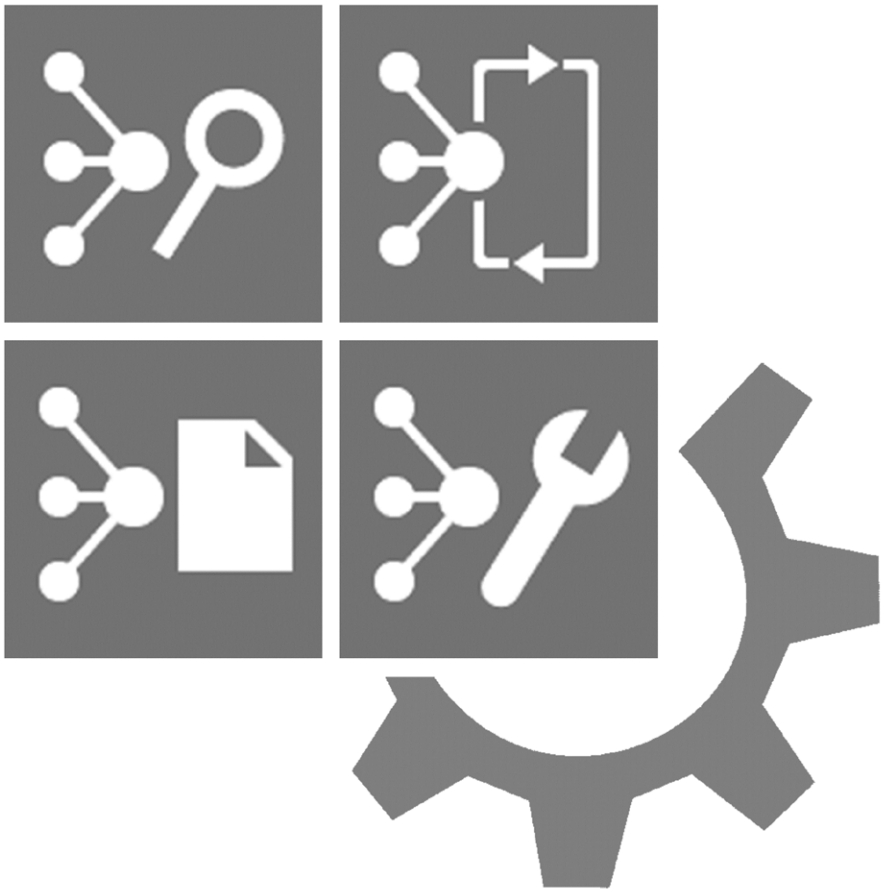
RENISHAW
apply innovation™

DataHUB Entwicklerhandbuch



Leere Seite

RENISHAW
apply innovation™

Inhalt

1 Bevor Sie beginnen

1.1 Geschäftsbedingungen und Gewährleistung

Sofern nicht zwischen Ihnen und Renishaw im Rahmen einer separaten, unterzeichneten schriftlichen Vereinbarung etwas anderes vereinbart wurde, werden die Ausrüstung und/oder Software gemäß den allgemeinen Geschäftsbedingungen von Renishaw verkauft, die Sie zusammen mit dieser Ausrüstung und/oder Software erhalten oder auf Anfrage bei Ihrer lokalen Renishaw Niederlassung erhältlich sind.

Renishaw übernimmt für seine Ausrüstung und Software für einen begrenzten Zeitraum (laut den allgemeinen Geschäftsbedingungen) die Gewährleistung, vorausgesetzt sie werden exakt entsprechend der von Renishaw erstellten verbundenen Dokumentation installiert und verwendet. Die genauen Angaben zur Gewährleistung sind in den allgemeinen Geschäftsbedingungen enthalten.

Ausrüstung und/oder Software, die Sie von einer Drittfirma erwerben, unterliegt separaten allgemeinen Geschäftsbedingungen, die Sie zusammen mit dieser Ausrüstung und/oder Software erhalten. Einzelheiten dazu erfahren Sie bei Ihrem Lieferanten.

1.2 Technische Änderungen

Renishaw behält sich das Recht vor, technische Änderungen ohne Vorankündigung vorzunehmen.

1.3 Patente

Merkmale der additiven Fertigungssysteme und ähnlicher Systeme von Renishaw sind Gegenstand eines oder mehrerer der folgenden Patente bzw. Patentanmeldungen.

1.3.1 RenAM 500 Serie (Q-, S- und Flex-Modelle)

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 Einführung

2.1 Lieferumfang

Dieses Dokument ist eine Anleitung zum Erstellen, Testen und Bereitstellen einer Plug-in-Softwarekomponente für die DataHUB Software. Im DataHUB-Benutzerhandbuch (Renishaw-Artikelnummer H-5800-6852) finden Sie einen Überblick und eine Erläuterung der Funktionen der DataHUB Software. Der Plug-in-Entwicklungszyklus beginnt mit dem Installieren von DataHUB auf dem Entwicklungs-PC. Danach wird mit Visual Studio eine .NET-Assembly mit dem Code für das Plug-in erstellt. Testen und debuggen Sie das Plug-in anschließend mit der entsprechenden Visual Studio-Debugging-Konfiguration, um dessen korrekte Funktionsweise zu verifizieren, bevor es auf dem Rechner, der in einem AM-Produktionssystem zur Datenerfassung dient (Data Collector PC bzw. DCPC) bereitgestellt wird.

2.2 Abkürzungen

Bezeichnung	Definition
AM	Additive Fertigung
AMPM	Überwachung des additiven Fertigungsprozesses
DCPC	Data Collection Personal Computer (der PC, der zur Erfassung der Daten genutzt wird)
HMI (MMS)	Mensch-Maschine-Schnittstelle (Touchscreen)
OEM	Original Equipment Manufacturer (Erstausrüster)
PC	Personal Computer
SPS	Speicherprogrammierbare Steuerung
REACH	Registrierung, Bewertung, Zulassung und Beschränkung von Chemikalien
WEEE-Richtlinie	Richtlinie zur Entsorgung von Elektro- und elektronischen Altgeräten

2.3 Sicherheitshinweise in diesem Benutzerhandbuch

Zusätzliche Informationen in diesem Benutzerhandbuch, die unbedingt gelesen und verstanden werden sollten, werden durch die Begriffe „Warnhinweis“, „Vorsichtshinweis“ oder „Hinweis“ hervorgehoben. Die nachstehenden Beispiele illustrieren dies eingehender.

2.3.1 Warnhinweise

Beispiel für einen Warnhinweis:

WARNHINWEIS: Ein Warnhinweis macht den Endbenutzer darauf aufmerksam, dass Verletzungsgefahr für ihn selbst oder andere Personen in der Nähe besteht, wenn den beschriebenen Verfahrensanweisungen nicht Folge geleistet wird.

2.3.2 Vorsichtshinweise

Beispiel für einen Vorsichtshinweis dieser Art:

VORSICHTSHINWEIS: Der Vorsichtshinweis macht den Endbenutzer darauf aufmerksam, dass eine Beschädigung der Ausrüstung möglich ist, wenn den Verfahrensanweisungen nicht Folge geleistet wird.

2.3.3 Hinweis

Beispiel für einen Hinweis dieser Art:

HINWEIS: Ein Hinweis liefert dem Endbenutzer wichtige Informationen oder hilft diesem bei der Ausführung der gerade zu erledigenden Aufgabe oder Tätigkeit.

2.4 Schulungsprogramm

Renishaw bietet eine Grundlagenschulung zur sicheren Nutzung von DataHUB. Darüber hinaus bietet Renishaw auch fortgeschrittene Schulungskurse für Bediener und Prozessingenieure an. Weitere Informationen finden Sie im DataHUB-Benutzerhandbuch (Renishaw Art. Nr. H-5800-6852) und dem Handbuch, das Sie im Rahmen des Schulungskurses erhalten, den alle Bediener vor der Nutzung von DataHUB absolvieren sollten.

2.5 Referenzdokumente

Neben diesem Benutzerhandbuch finden Sie in den folgenden Dokumenten zusätzliche Informationen zu weiteren Aspekten der InfiniAM® Suite und des Renishaw AM-Systems.

- *RenAM 500Q/S Additives Fertigungssystem* Installationshandbuch (Renishaw Art. Nr. H-5800-4345)
- *RenAM 500Q/S Additives Fertigungssystem* Benutzerhandbuch (Renishaw Art. Nr. H-5800-4346)
- *InfiniAM® und DataHUB Software* Installationshandbuch (Renishaw Art. Nr. H-5800-6844)
- *DataHUB* Benutzerhandbuch (Renishaw Art. Nr. H-5800-6852)
- *InfiniAM® Spectral* Benutzerhandbuch (Renishaw Art. Nr. H-5800-6840)
- *InfiniAM® Camera* Benutzerhandbuch (Renishaw Art. Nr. H-5800-6848)
- Renishaw Licence Manager v2.x Benutzerhandbuch (Download über den Link in der InfiniAM Lizenz-E-Mail)
- Export Packets Plug-in (Plug-in für den Paketexport) (Renishaw Art. Nr. 5800-6788)

3 Ersatzteile

DataHUB enthält keine vom Benutzer zu wartenden Teile. Bei einem Ausfall oder Versagen muss die Software zur Wiederherstellung erneut installiert und konfiguriert werden.

Vereinbaren Sie hierzu einen Servicetermin mit Ihrer lokalen Renishaw-Niederlassung. Die Kontaktangaben finden Sie im Abschnitt 4, „Kontaktangaben“.

InfiniAM und DataHUB Software werden periodisch aktualisiert. Alle Abonnenten sind berechtigt, die neuesten Software-Releases über ihr Konto auf MyRenishaw.com herunterzuladen.

Leere Seite

4 Kontaktangaben

Telefonnummer	+49 (0) 7127 9810	
Geschäftszeiten	Montag bis Donnerstag	08:00 bis 17:00 Uhr (UTC und DST)
	Freitag	08:00 bis 16:00 Uhr (UTC und DST)
E-Mail	Support-AM-DE@renishaw.com	
Kundendienst	Renishaw GmbH Karl-Benz Straße 12 72124 Pliezhausen Deutschland	
AM-Systemtyp		
Seriennummer des AM-Systems		
Versionsnummern der Software	MMS Revision	
	SPS Revision	
	PC Revision	
Seriennummer der InfiniAM Spectral Hardware (MeltVIEW Modul)		
Versionsnummer der InfiniAM Software		
Versionsnummer der DataHUB Software		

Bitte geben Sie bei Ihrer Anfrage die obigen Informationen an. Die Daten des jeweiligen Renishaw AM-Systems sind auf dem Typenschild auf der Rückseite des Systems zu finden. Angaben über das MeltVIEW Modul befinden sich auf dem Typenschild, das bei der Installation von InfiniAM auf dem Renishaw AM-System einsehbar ist. Die Daten der CameraVIEW Hardware sind auf einem Aufkleber auf der Rückseite der Kamera zu finden. Die CameraVIEW Hardware befindet sich hinter einer Abdeckung über der Kammer.

Für zusätzliche Unterstützung nehmen Sie bitte mit Ihrer örtlichen Renishaw-Niederlassung Kontakt auf. Siehe:

www.renishaw.de/Renishaw-Weltweit

Leere Seite

5 Sicherheitshinweise

5.1 Einführung

WARNHINWEIS: Sofern in diesem Dokument nicht anders angegeben, stimmen sämtliche Sicherheitshinweise mit der Bedienungsanleitung und dem Installationshandbuch für das betreffende Renishaw AM-System überein.

WARNHINWEIS: Stellen Sie sicher, dass eine Abdeckplatte oder das MeltVIEW Hardwaremodul an der Laseröffnung angebracht ist, bevor Sie den Laser des AM-Systems einschalten.

WARNHINWEIS: Das MeltVIEW Hardwaremodul ist nicht in den Lasersicherheitsschaltkreis integriert. Wird der Laser gezündet, während das MeltVIEW Hardwaremodul nicht angebracht ist, kann schädliches Laserlicht aus der Laseröffnung des AM-Systems austreten.

5.2 Spezifische Laser- und Warnhinweise für DataHUB Generator

Die Installation des DataHUB Generators ist nicht mit zusätzlichen Sicherheits- oder Warnschildern für das AM-System verbunden.

Leere Seite

6 Betrieb

6.1 Einführung

Durch seine Plug-in-Architektur unterstützt das InfiniAM Softwarepaket die Echtzeitanalyse von Prozessüberwachungsdaten, die von den LaserVIEW und MeltVIEW Hardware-Plattformen erfasst werden. Ein Plug-in ist eine eigenständige Softwareeinheit, die auf einem zur Datenfassung genutzten PC (DCPC) installiert wird, auf dem DataHUB läuft. Bei der Verarbeitung der gesammelten Daten für einen additiven Fertigungsprozess leitet DataHUB die Prozessüberwachungsdaten an die Plug-ins weiter, sodass diese eine maßgeschneiderte Analyse durchführen können. DataHUB unterstützt mehrere Plug-ins gleichzeitig. Dadurch können verschiedene Analysealgorithmen und -techniken auf die Prozessüberwachungsdaten angewandt werden. Durch die Integration in das Renishaw Central System können Plug-ins die Analyseergebnisse zusammen mit den Messwerten anderer Maschinensensoren, die während des Baus erfasst wurden, darstellen.

DataHUB unterstützt zwei Plug-in API-Versionen: Version 1.0 (V1.0) unterstützt Plug-ins, die als Pakete zusammengefasste Baudaten verarbeiten (siehe „Samples in Paketen“ auf Seite 6-8), Version 2 (V2.0) unterstützt Plug-ins, die Baudaten als Token-Streams verarbeiten (siehe „Samples in Token-Stream“ auf Seite 6-8). Dabei ist zu beachten, dass die V2.0-API die V1.0-API nicht ersetzt; beide geben die gleichen Überwachungsdaten – wenn auch in unterschiedlichen Formaten – an die Plug-ins weiter. Die zu verwendende API sollte unter Berücksichtigung der durchzuführenden Analyse ausgewählt werden: V1.0 liefert wesentlich weniger Pakete als die von V2.0 gelieferten Messergebnisse (Samples). Dies kann, falls feine Details der Sample-Daten nicht für jede Analyseaufgabe erforderlich sind, besser geeignet sein.

Dieses Dokument beschreibt die Architektur des Plug-in-Systems, Phasen im Leben eines Plug-ins sowie weitere Aspekte des Systems, die für beide API-Versionen zutreffen.

Das Export Packets Plug-in für den Export der Pakete erstellt eine tabulatorgetrennte Datei mit einer Zusammenfassung der Prozessüberwachungsdaten, die von der LaserVIEW- und MeltVIEW-Hardware während eines Baus erfasst werden. Dieses Plug-in wird zusammen mit DataHUB geliefert und erfordert keine zusätzliche Lizenz.

6.2 Architektur

Abbildung 1 gibt einen groben Überblick der Beziehungen zwischen den Hardware- und Softwareelementen des Systems:

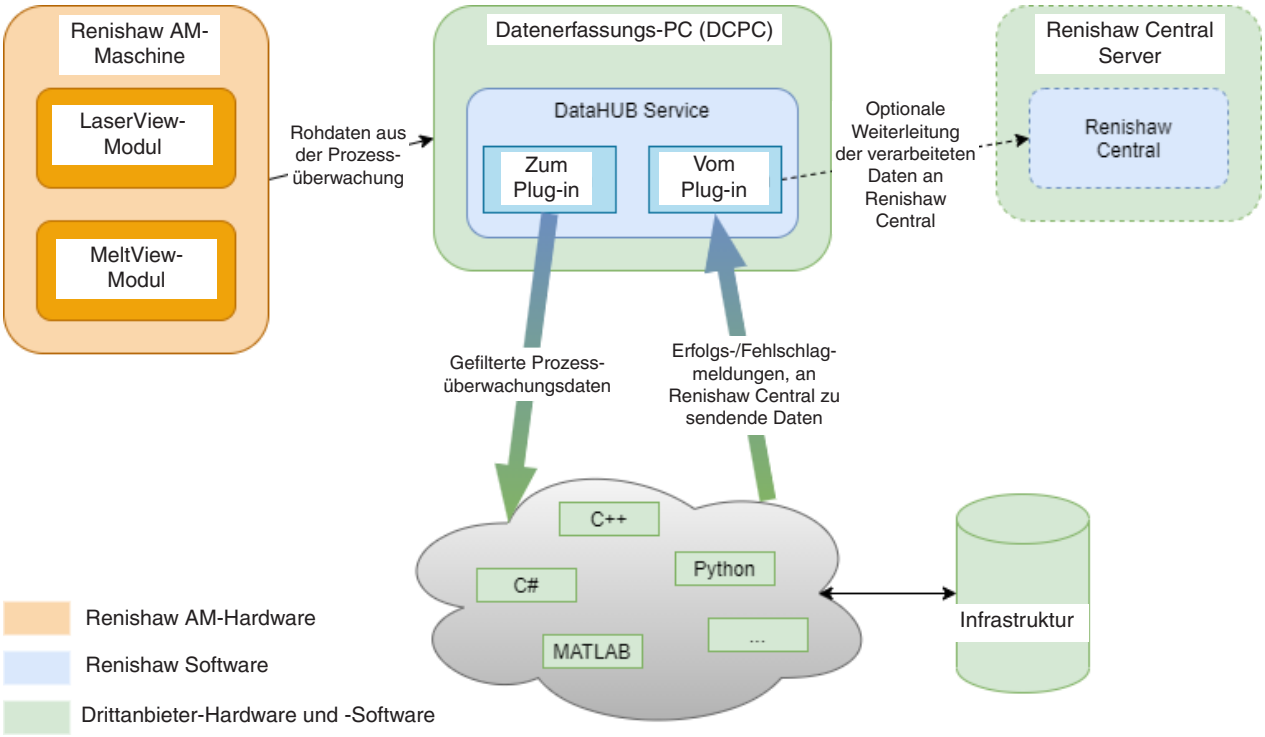


Abbildung 1 Die DataHUB Plug-in-Architektur

6.2.1 Datenfluss

Wenn eine mit LaserVIEW- und MeltVIEW-Hardware ausgestattete Renishaw AM-Maschine einen Bauvorgang durchführt, werden die Prozessüberwachungsdaten an einen Datenerfassungs-PC (DCPC) übertragen. Diese Daten werden vom DataHUB Service, der auf diesem DCPC läuft, an alle installierten Plug-ins weitergeleitet. DataHUB führt pro Bau eine Instanz jedes installierten Plug-ins aus. Jeder Instanz wird ein eindeutig benannter Ausgabeordner für alle gespeicherten Ausgaben zugewiesen, und jede Instanz wird unabhängig ausgeführt, um sicherzustellen, dass ein Ausfall in einer Instanz nicht zu einem generellen Systemversagen führt.

Abbildung 2 zeigt den typischen Datenfluss bei der Verarbeitung eines Baus

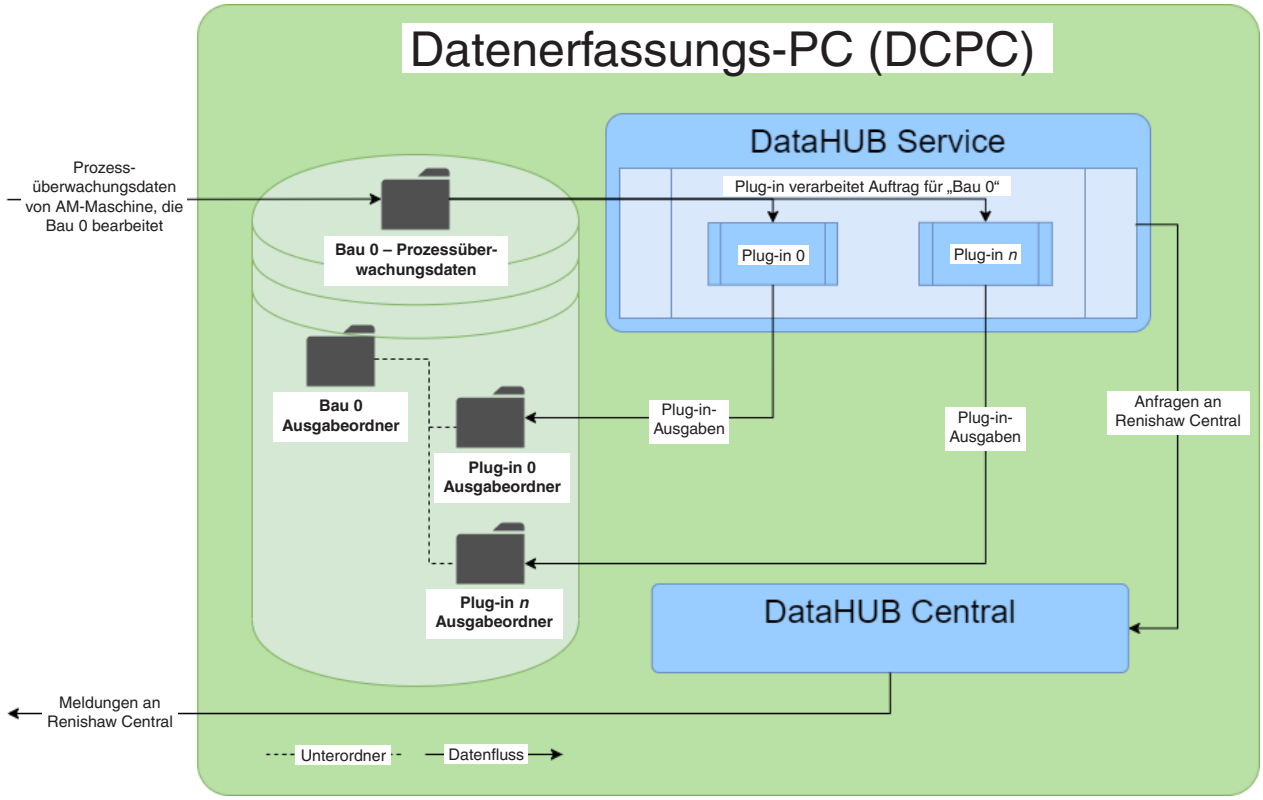


Abbildung 2 Baudatenflüsse

Die von einem AM-Rechner, auf dem ein Bau ausgeführt wird, erfassten Prozessüberwachungsdaten werden an einen Ordner auf dem DCPC gesendet. Der DataHUB Service überwacht diesen Ordner und sendet neue Daten nach ihrem Eintreffen an die mit dem Bauauftrag verbundenen Plug-in-Instanzen. Die Plug-ins melden dann dem DataHUB Service gegebenenfalls Ergebnisse (Zeitreihendaten, Alarme und Analyseergebnisse), die an Renishaw Central gesendet werden sollen. Diese Upload-Anforderungen werden vom DataHUB Central Service verwaltet.

6.2.2 Plug-in-Analysetechnik

DataHUB ist weitgehend agnostisch gegenüber der zur Implementierung der Plug-ins verwendeten Technologie und der erforderlichen unterstützenden Infrastruktur. DataHUB stellt die folgenden beiden Anforderungen:

1. Das Plug-in ist eine .NET-Assembly, die eine Reihe von vorgegebenen Methoden (die Plug-in-API) ausführt,
2. Alle Laufzeit-abhängigen Elemente werden auf dem DCPC installiert.

DataHUB ruft die Plug-in-API-Methoden mit Werten auf, die aus den Prozessüberwachungsdaten übernommen werden. Der Plug-in-Implementierung steht es dann frei, diese Daten der Technologie zuzuführen, die für die jeweilige Analyseaufgabe am besten geeignet ist. Eine detaillierte Erläuterung, wie man .NET mit anderen Technologien und Programmiersprachen integriert, würde den Rahmen dieses Dokuments sprengen, darüber ist jedoch eine Fülle von öffentlich zugänglichen Informationen verfügbar.

6.2.3 Der „Push“-Datenfluss

Der DataHUB Service „schiebt“ Daten zu den Plug-ins; die Plug-in-APIs sind nicht zur Abfrage von Datensätzen und Extraktion von Datenwerten angelegt. Der typische Anwendungsfall, bei dem die Analyse in Echtzeit durchgeführt wird, während der Bau voranschreitet, wird durch diese Form von Datenfluss am besten unterstützt. Um archivierte Datensätze zusätzlichen Plug-in-Analysen zu unterziehen, können Sie den DataHUB Service über den DataHUB Generator anweisen, die Daten erneut zu verarbeiten.

6.2.4 Die Phasen im Leben eines Plug-ins

Damit ein Plug-in von DataHUB verwendet werden kann, wird es zunächst anhand der Implementierungsanforderungen der einzelnen APIs validiert. Erst wenn es für konform befunden wurde, werden Instanzen erstellt und zur Analyse von Baudaten verwendet. Nach der Validierung wird eine Instanz des Plug-ins für jede neue Aufgabe zur Verarbeitung von Baudaten erstellt. Die Phasen im Leben eines Plug-ins sind in Abbildung 3 dargestellt:

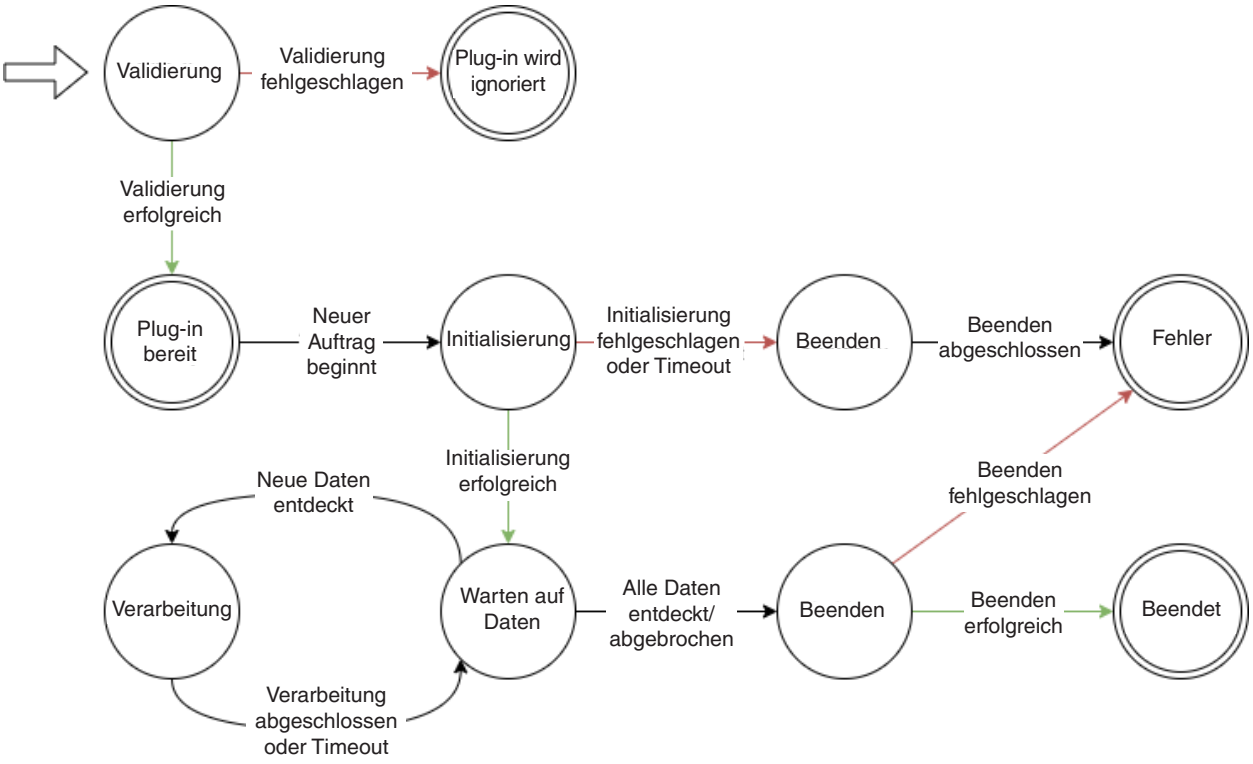


Abbildung 3 Die verschiedenen Zustände, die ein Plug-in während seiner Lebensdauer durchlaufen kann

6.2.4.1 Validierung

Beim Start von DataHUB wird nach .NET-Assemblies gesucht, die eine oder mehrere öffentliche Klassen implementieren, die mit den DataHUB Plug-in-APIs konform sind. Alle konformen Assemblies werden als gültige Plug-ins registriert und instanziiert, sobald ein Auftrag unter Verwendung von **Process LaserVIEW and MeltVIEW via Plug-ins** (LaserVIEW und MeltVIEW mit Plug-in verarbeiten) gestartet wird.

Auf Wunsch kann eine Assembly mehrere Plug-ins umfassen und verschiedene Versionen der API nutzen.

6.2.4.2 Initialisierung

Wenn DataHUB einen Plug-in-Auftrag startet, wird eine neue Instanz eines Plug-ins erstellt und die zugehörige *Initialise*-Methode zur Initialisierung aufgerufen. Die Werte, die an diese Methode übermittelt werden, sind „Baukonstanten“, z. B. die Anzahl der Laser zur Berechnung des Ressourcenbedarfs durch das Plug-in usw. Die Methode meldet zurück, ob die Initialisierung erfolgreich war oder fehlgeschlagen ist. Wenn sie einen Fehler ausgibt, wird sofort die Methode *Shutdown* (Beenden) für die Instanz aufgerufen. Das Plug-in kann optional Definitionen für Zeitreihenachsen zurückgeben, die mit Datenpunkten aus der nachfolgenden Verarbeitung befüllt werden (siehe „Weiterleitung von Daten an Renishaw Central“ auf Seite 6-6).

6.2.4.3 Warten auf Daten/Verarbeitung

Die Datenverarbeitungsmethode wird so oft aufgerufen, wie es notwendig ist, um alle Prozessüberwachungsdaten eines Baus an die Plug-in-Instanz zu übergeben. Bei jedem Aufruf meldet das Plug-in, ob es die Daten erfolgreich verarbeitet hat oder ob die Verarbeitung fehlgeschlagen ist. Bei einem Fehler geht das Plug-in nicht in einen Fehlerzustand über, sondern nimmt weiterhin Daten entgegen (es wird davon ausgegangen, dass die Daten anderer Schichten für das Plug-in dennoch relevant sind). Das Plug-in kann optional Zeitseriendatenpunkte, Warnungen und andere Analyseergebnisse zurückgeben, die an Renishaw Central weitergeleitet werden (siehe „Weiterleitung von Daten an Renishaw Central“).

6.2.4.4 Beenden

Wenn keine weiteren Daten zu verarbeiten sind (d. h., der Bau abgeschlossen oder abgebrochen wurde), wird die *Shutdown-Methode* aufgerufen. Diese Methode veranlasst alle abschließenden Verarbeitungen an den Baudaten, die Freigabe aller übernommenen Ressourcen usw.

6.2.5 Zeit

Die Zeit wird über die API in Mikrosekunden seit dem Baubeginn einer Schicht mitgeteilt. Die Zeitangabe 0 signalisiert den Beginn einer Schicht, die Zeit 1000 ist 1000 Mikrosekunden nach dem Beginn der Schicht usw. Falls erforderlich (z. B. bei der Übermittlung an Renishaw Central) übersetzt DataHUB alle von einem Plug-in ausgegebenen relativen Zeitangaben in einen absoluten Zeitrahmen wie UTC.

6.2.6 Weiterleitung von Daten an Renishaw Central

Wenn DataHUB mit Renishaw Central verbunden ist, können die verknüpften Plug-ins ihre Analyseergebnisse mittels DataHUB weiterleiten. Wenn die von den Methoden *Initialise* und *Process* zurückgegebene Zeichenfolge dem in den API-Definitionen aufgezeichneten JSON-Schema entspricht (siehe Abschnitt 6.4, „API V2 Plug-in“), sendet DataHUB die beinhalteten Daten an den Renishaw Central Endpunkt.

DataHUB ist in der Lage, die folgenden drei Arten von Uploads an Renishaw Central zu leiten:

- Meldungen
- Analyseergebnisse
- Zeitreihen

6.2.6.1 Meldungen

Eine Meldung zeigt an, dass zu einem bestimmten Zeitpunkt ein wichtiges Ereignis eingetreten ist. Meldungen können von einem Plug-in generiert werden, wenn dieses eine Anomalie in den Baudaten feststellt, wenn die Daten beispielsweise darauf hindeuten, dass ein Baufehler aufgetreten ist oder bevorsteht. Es gibt drei Meldungsstufen: *Info*, *Warnung* und *Fehler*. Sie dienen dazu, der verknüpften Anwendung (z. B. der Renishaw Central Website) mitzuteilen, wie diese Meldung einzustufen ist.

Meldungen können nur in der Phase *Process* (Verarbeiten) ausgelöst werden.

6.2.6.2 Analyseergebnisse

Neben Meldungen können auch Ergebnisse der Baudatenanalyse zurückgegeben werden. Sie haben eine flexible Definition, sodass ihr Inhalt je nach Ergebnistyp spezialisiert werden kann. Falls also (zum Beispiel) die Renishaw Central Website möglicherweise nicht alle Facetten eines Analyseergebnisses darstellen kann, könnte eine benutzerdefinierte nachgelagerte Anwendung geschrieben werden, die dazu in der Lage ist.

Analyseergebnisse können nur während der Phase *Process* (Verarbeiten) gemeldet werden.

6.2.6.3 Zeitreihen

Eine Zeitreihe stellt eine Reihe von Werten im Zeitverlauf dar. Zeitreihen können Daten jeder Art enthalten, vorausgesetzt, sie können im Zeitverlauf erfasst werden. Während der *Verarbeitungsphase* können Plug-ins zwei Arten von Daten zu einer Zeitreihe hinzufügen: Werte und Grenzwerte. Werte sind Zeit-/Wert-Paare, die der Zeitreihe einen Datenpunkt hinzufügen. Grenzwerte definieren erwartete Bereiche, um die Anzeige zu erleichtern und Extremwerte aufzuzeigen.

Zeitreihen unterscheiden sich von Warnmeldungen und Analyseergebnissen dadurch, dass sie zunächst definiert werden müssen, bevor Datenpunkte hinzugefügt werden können. Plug-ins können Zeitreihendefinitionen über ihre *Initialise*-Methode ausgeben. Sollte das Plug-in zu einem späteren Zeitpunkt versuchen, Daten zu einer Zeitreihe hinzuzufügen, die nicht in dieser ursprünglichen Definitionsliste enthalten war (oder falls die Definitionen selbst auf irgendeine Weise ungültig waren), werden die Daten zurückgewiesen und gehen verloren.

6.2.7 Audit-Protokolle

DataHUB legt eine Reihe von Protokollen an, um die durchgeführten Vorgänge aufzuzeichnen. Einige dieser Protokolle können zur Diagnose von Plug-in-Problemen verwendet werden. Die Protokolle befinden sich im folgenden Ordner:

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

Protokolle werden nach dem Tag benannt, an dem sie aufgezeichnet wurden, und zwar im Format „yyyyMMdd“.

Die wohl wichtigsten Protokolleinträge für Plug-in-Entwickler sind:

- Beim Start zeichnet DataHUB auf:
  - die Versionsnummern
  - die Lizenzdetails
  - die Validierung der Plug-in-Assemblies

- Beim Start eines Baus zeichnet DataHUB auf:
  - das Erstellen von Plugin-Instanzen
  - die Werte, die zur Initialisierung jeder Plug-in-Instanz verwendet werden
  - das Ergebnis der Methode *Initialise*

- Während eines Baus zeichnet DataHUB auf:
  - nicht-triviale Ergebnisse von *Verarbeitungsaufrufen* des Plug-ins (da Plug-ins triviale Ergebnisse melden können, die nicht protokolliert zu werden brauchen, können auf diese Weise unnötige Protokolleinträge vermieden werden)

- Nach Beendigung eines Baus zeichnet DataHUB auf:
  - das Ergebnis jedes Aufrufs der *Shutdown-Methode* des Plug-ins

6.2.8 Vorverarbeitung von Prozessüberwachungsdaten

Die LaserVIEW- und MeltVIEW-Module sammeln Daten in Form eines Stroms von Messergebnissen (Samples), die jeweils mit einem bestimmten Laser, einer Position auf dem Pulverbett und verschiedenen Sensormesswerten zu einem bestimmten Zeitpunkt verbunden sind. DataHUB unterzieht diese „rohen“ Prozessüberwachungsdaten einer Vorverarbeitung. Bei den Plug-ins V1.0 fasst DataHUB die Samples zu Paketen zusammen, bei V2.0 werden die Sample-Daten zu einem Token-Stream zusammengefasst.

6.2.8.1 Samples in Paketen

Der DataHUB Service fasst aufeinanderfolgende Samples, die beim Abfeuern des Lasers in der gleichen Position sind, zu einem Paket zusammen. Die Startzeit eines Pakets wird dem ersten beinhalteten Sample entnommen, die Dauer des Pakets wird anhand des letzten beinhalteten Samples berechnet. Die LaserVIEW- und MeltVIEW-Sensormesswerte der beinhalteten Samples werden gemittelt.

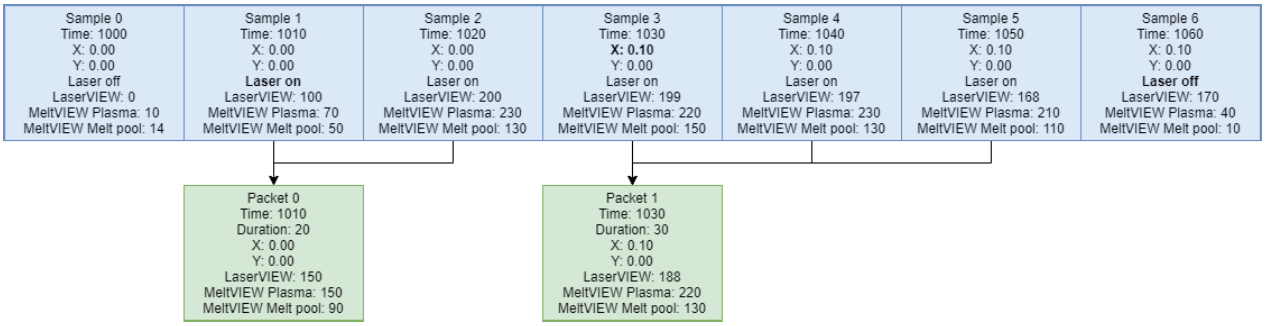


Abbildung 4 Samples in Paketen

Abbildung 4 zeigt sieben Samples, die in zwei Pakete quantisiert werden. Das erste Sample wird ignoriert, da der Laser hier nicht zündet. Da sie dieselbe Position haben und der Laser zündet, werden die Samples 1 und 2 in einem Paket zusammengefasst. Die Position von Sample 3 weicht vom vorausgehenden Sample ab. Daher beginnt hier ein neues Paket, das Samples aufnimmt, bis der Laser aufhört zu feuern. Obgleich es die gleiche Position hat wie Sample 5, wird das letzte Sample 6 nicht hinzugefügt, da der Laser jetzt ausgeschaltet ist.

6.2.8.2 Samples in Token-Stream

Bei der Konvertierung der Messergebnisse (Samples) aus der Prozessüberwachung in einen Token-Stream erstellt DataHUB eine Reihe von *Sample*-Tokens, die von anderen Informationstoken umgeben sind, und gemeinsam die Struktur und den Inhalt des Baus beschreiben. Im Gegensatz zu V1.0 werden hier Samples, die erfasst wurden, während der Laser nicht feuerte, berücksichtigt und dem Token-Stream hinzugefügt.

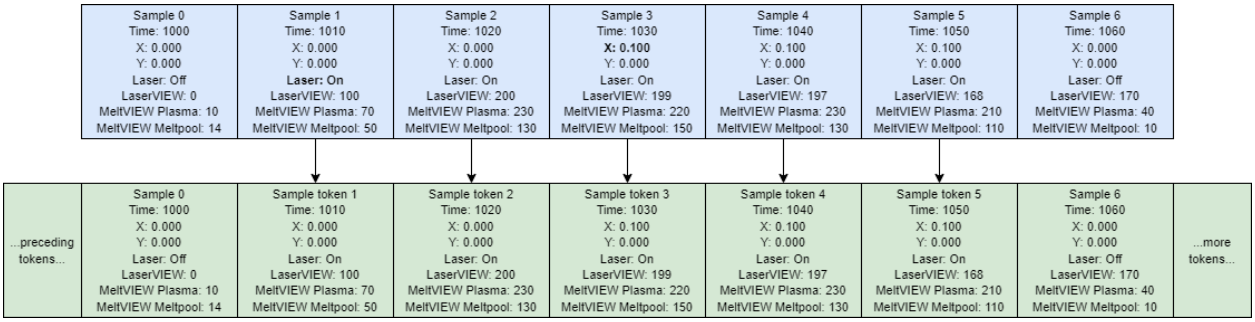


Abbildung 5 Samples in Token-Stream

Abbildung 5 zeigt einen Teil eines Token-Streams mit denselben Samples, die oben als *Samples in Paketen* zu sehen sind.

6.3 Das Plug-in für API V1

Dieses Handbuch ist eine Anleitung zum Erstellen, Testen und Bereitstellen einer Plug-in-Softwarekomponente V1.0 für die DataHUB Software. Der Plug-in-Entwicklungszyklus beginnt mit dem Installieren von DataHUB auf dem Entwicklungs-PC. Danach wird mit Visual Studio eine .NET-Assembly mit dem Code für das Plug-in erstellt. Testen und debuggen Sie das Plug-in anschließend mit der entsprechenden Visual Studio-Debugging-Konfiguration, um dessen korrekte Funktionsweise zu verifizieren, bevor es auf dem Rechner, der in einem AM-Produktionssystem zur Datenerfassung dient (Data Collector PC bzw. DCPC) bereitgestellt wird.

6.3.1 Voraussetzungen und vorbereitende Schritte

Lesen Sie in Verbindung mit diesem Handbuch auch die folgenden Dokumente:

- *InfiniAM® und DataHUB Software* Installationshandbuch (Renishaw Art. Nr. H-5800-6844)
- *DataHUB* Benutzerhandbuch (Renishaw Art. Nr. H-5800-6852)
- Benutzerhandbuch für Renishaw Licence Manager v2.x

Bevor Sie fortfahren, sollten Sie die folgenden vorbereitenden Schritte auf dem PC durchführen, der für die Plug-in-Entwicklung verwendet werden soll:

- Vergewissern Sie sich, dass der PC mit einem Grafikprozessor ausgestattet ist, der zumindest die DataHUB Mindestanforderungen erfüllt (siehe die Hardware-Spezifikationen für den Datenerfassungs-PC in den Handbüchern für *InfiniAM® und DataHUB*).
- Vergewissern Sie sich, dass die aktuellen Treiber für die GPU installiert sind.
- Prüfen Sie, dass der Lizenzserver über genügend geeignete Lizenzen für DataHUB und Spectral API verfügt.
- Überprüfen Sie den Netzwerkzugriff auf den Lizenzserver.
- Stellen Sie sicher, dass eine Version von Microsoft Visual Studio installiert ist, die das .NET Framework 4.7.2 unterstützt.

6.3.2 DataHUB für die Plug-in-Entwicklung installieren

Der DataHUB Service muss auf dem Entwicklungs-PC installiert werden. Im Laufe der Installation können Sie wählen, welche Lizenzen genutzt werden sollen. Vergewissern Sie sich, dass die Option **LaserVIEW/ MeltVIEW Pluggability** (LaserVIEW/MeltVIEW Plug-in-Kompatibilität) aktiviert ist.

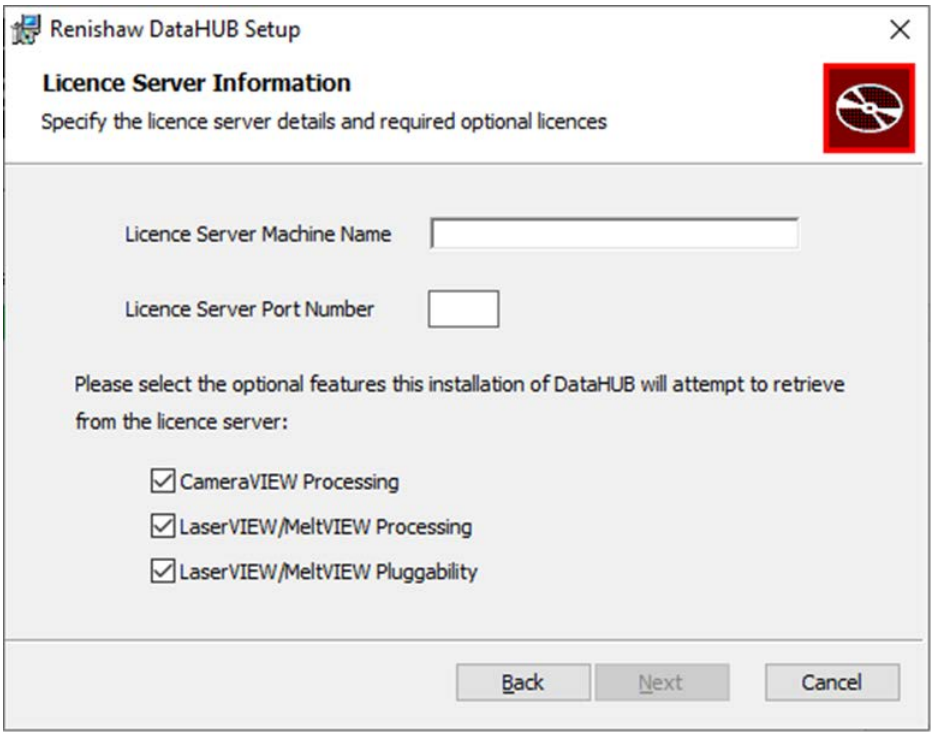


Abbildung 6 DataHUB Lizenzen

6.3.3 Implementierung des Plug-ins

Die nachfolgenden Abschnitte veranschaulichen den gesamten Arbeitsablauf beim Erstellen und Kompilieren, Debuggen und Bereitstellen eines Plug-ins mit Visual Studio.

Plug-ins werden in C# geschrieben und definieren eine öffentliche Klasse (public class), die den spezifischen Satz öffentlicher Methoden implementiert, anhand derer DataHUB beim Verarbeiten der Prozessüberwachungsdaten eines Baus Instanzen des Plug-ins identifiziert, validiert und verwendet.

HINWEIS: Die unten gezeigten Screenshots stammen aus Microsoft Visual Studio 2019, der Arbeitsablauf ist jedoch in anderen Versionen weitgehend ähnlich.

6.3.3.1 Visual Studio-Lösung erzeugen

Starten Sie Visual Studio und wählen Sie *Create a new project* (Neues Projekt erstellen). Wählen Sie unter den verfügbaren Optionen *Class Library (.NET Framework)* (Klassenbibliothek (.NET Framework)):

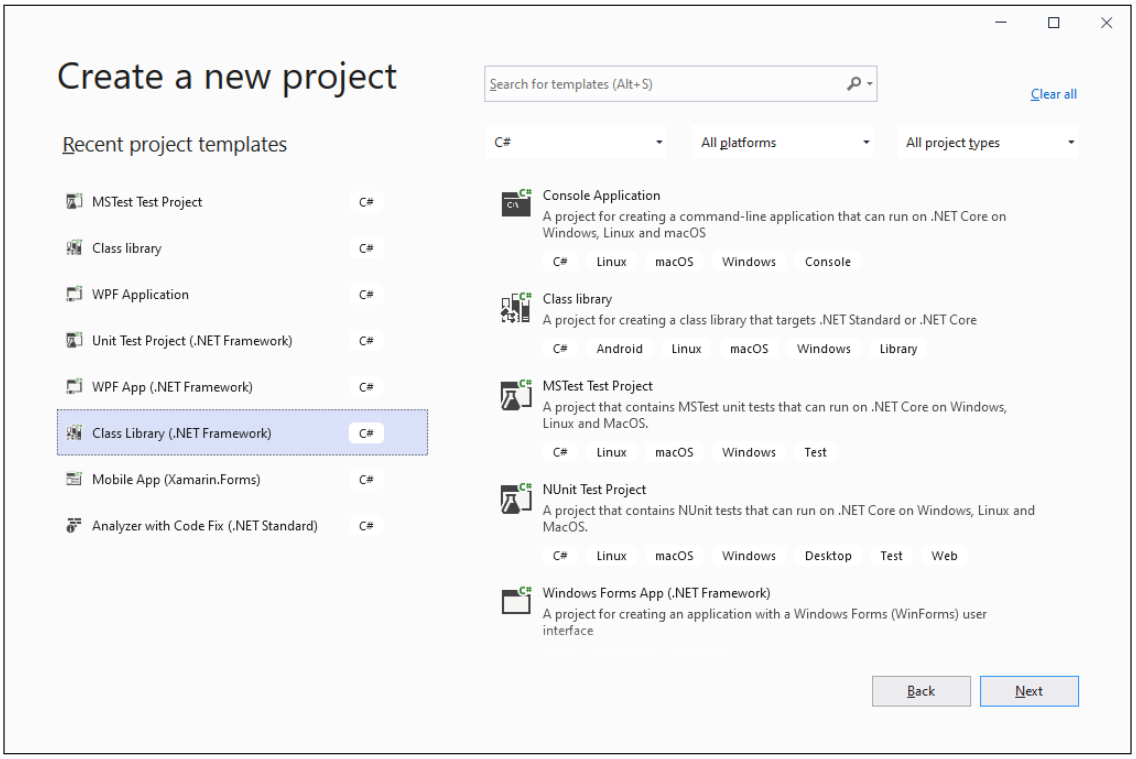


Abbildung 7 Lösung erstellen

Nun müssen Sie das Projekt konfigurieren:

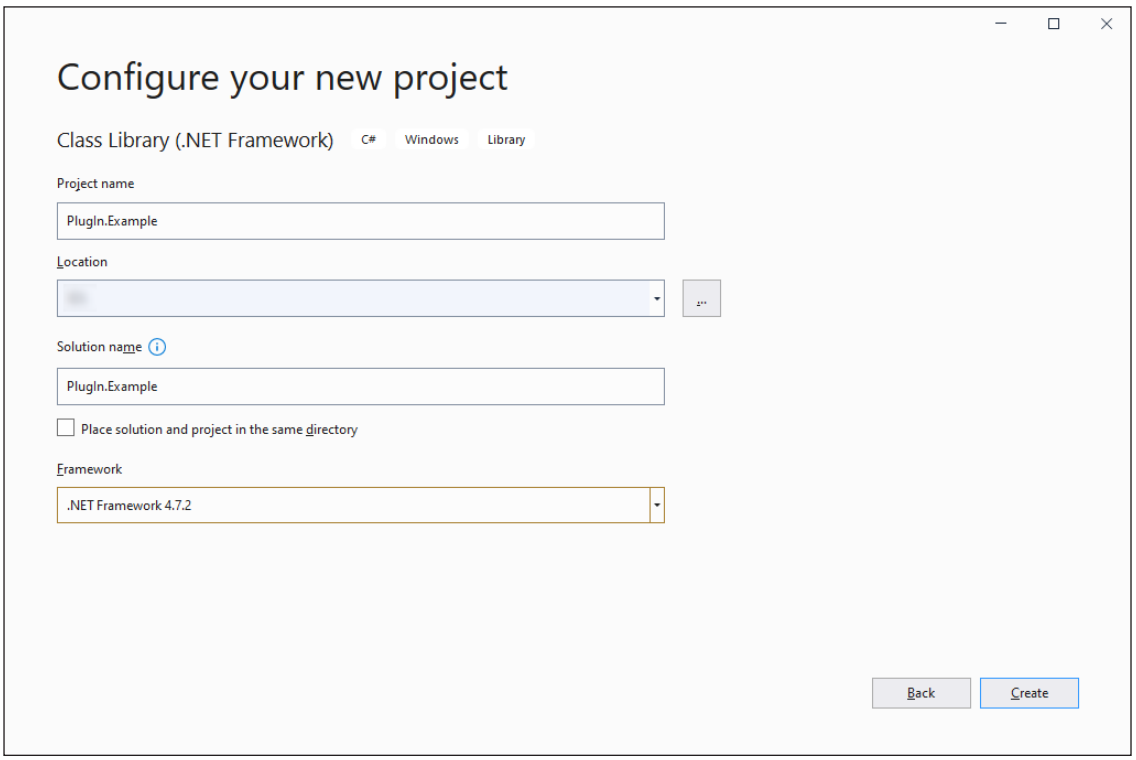


Abbildung 8 Projekt konfigurieren

DataHUB erfordert, dass eine Plug-in-Assembly mit **PlugIn.** beginnt (das Wort „PlugIn“ gefolgt von einem Punkt). Wählen Sie unter *Location* (Speicherort), wo das Projekt gespeichert werden soll, und erzeugen Sie das Plug-in.

HINWEIS: DataHUB unterstützt Plug-ins, die auf x86/x64/Any CPU ausgerichtet sind. Als Ziel wird Framework 4.7.2 oder höher empfohlen.

6.3.3.2 Plug-in-Typ benennen

Die Standard-Quelldatei Class1.cs enthält die Definition von `Class1`, die wir als Ausgangspunkt verwenden.

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

Sobald DataHUB eine potenzielle Plugin-Assembly entdeckt hat, sucht es nach einem `PlugIn` des Typs „Public“. Benennen Sie folglich `Class1` in `PlugIn` um.

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

6.3.3.3 API hinzufügen

DataHUB verlangt, dass potenzielle Plug-in-Typen die folgenden Public-Methoden implementieren:

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

Fügen Sie der Klasse die folgenden Methodenimplementierungen hinzu:

```
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() => "";
}
```

Beachten Sie, dass alle API-Methoden einen `string` zurückgeben (der für den Moment leer gelassen wird). Der Inhalt dieser zurückgegebenen Zeichenfolge signalisiert DataHUB das Ergebnis der Methode. Die Rolle des Rückgabewerts wird nachfolgend bei der eingehenden Erläuterung der einzelnen Methoden näher erklärt.

6.3.3.4 Die *APIVersion*-Methode

Diese Methode meldet DataHUB die Version der API, die das Plug-in verwendet. Für die API-Version 1.0 muss sie das Literal 1.0 zurückgeben:

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

6.3.3.5 Die *DisplayName*-Methode

Die Zeichenkette, die diese Methode zurückgibt, enthält den Namen, der als Plug-in-Name im DataHUB Monitor und bei der Erstellung des Ausgabeordners für das Plug-in angezeigt wird. Dieser Name muss zwar nicht zwingend eindeutig sein, erleichtert jedoch das Identifizieren des Plug-ins, wenn z. B. mehrere Versionen auf einem DCPC installiert sind.

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

6.3.3.6 Die *Initialise*-Methode

DataHUB ruft diese Methode zu Beginn eines Baus auf, bevor irgendwelche Schichtdaten an das Plug-in gesendet werden. Sie liefert dem Plug-in folglich *bauinvariante Daten*. In unserem Beispiel-Plug-in weist die *Initialise*-Methode dem Feld **_outputFolderPath** den Pfad zum Ausgabeordner der Instanz zu:

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

6.3.3.7 Die ProcessPackets-Methode

DataHUB ruft diese Methode auf, nachdem alle Daten für eine Schicht empfangen wurden. Im Beispiel-Plug-in erstellt die *ProcessPackets*-Methode zunächst eine Zeile mit durch Trennzeichen separierten Spaltenüberschriften und fügt dann die Werte für die Daten der Prozessüberwachungspakete Zeile für Zeile hinzu.

```
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 Die *Shutdown*-Methode

DataHUB ruft diese Methode auf, nachdem alle Daten für einen Bau an das Plug-in gesendet wurden oder wenn ein vorheriger Plug-in-Aufruf fehlgeschlagen ist. Die Methode hat keine Parameter.

Diese Methode signalisiert, dass das Plug-in sämtliche abschließenden Verarbeitungen an den Baudaten vornehmen, alle eventuell übernommenen Ressourcen freigeben soll usw. In unserem Beispiel-Plug-in lautet die *Shutdown*-Methode:

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

6.3.4 Plug-in testen und debuggen

Erstellen Sie die Lösung mit *Debug*, öffnen Sie einen Dateibrowser und suchen Sie den Ausgabeordner, der eine *.dll*-Datei mit dem Namen des Projekts enthält, zum Beispiel *PlugIn.Example.dll*. (Zusatzdateien wie Debug-Symbole usw. können ebenfalls vorhanden sein).

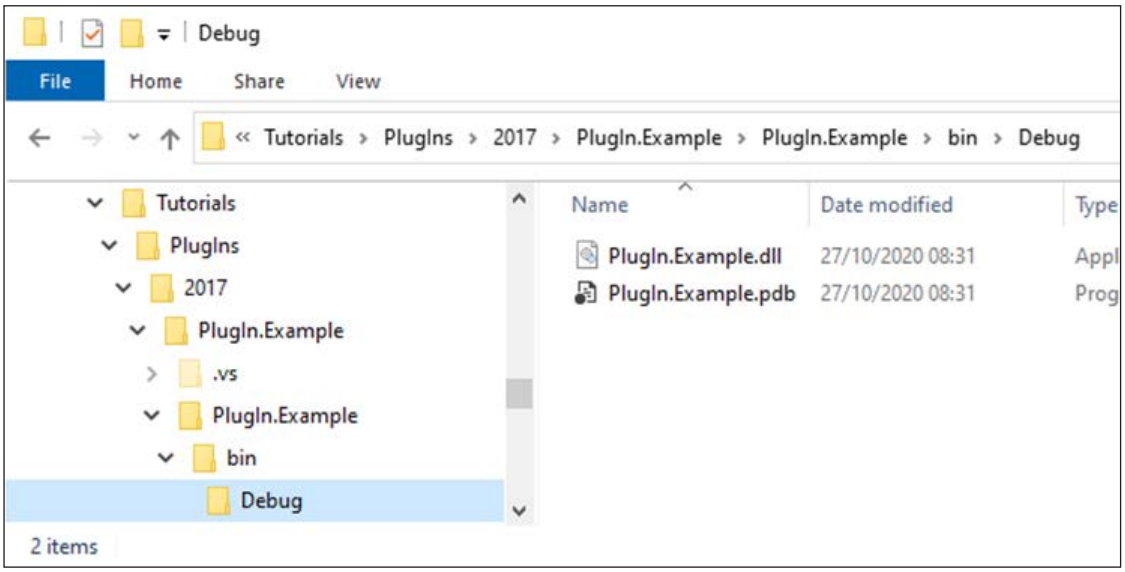


Abbildung 9 Plug-in-Binärdateien

Kopieren Sie diesen Ordner und fügen Sie ihn in den Unterordner *PlugIns* des Datenordners von DataHUB ein:

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

HINWEIS: Normalerweise befindet sich $PROGRAMDATA$ unter <C:\ProgramData>, der Ordner kann aber auch ausgeblendet sein. Aktivieren Sie in diesem Fall die Option **View, Show/hide, Hidden items** (Ansicht, Erweiterte Einstellungen, Ausgeblendete Dateien, Ordner und Laufwerke anzeigen) im Windows Datei-Explorer.

Um den Inhalt dieses Ordners zu ändern, sind möglicherweise Administratorrechte erforderlich.

Benennen Sie den Ordner *Debug* um, sodass er dem Namen des Plug-ins entspricht. In der nachfolgenden Abbildung wurde der Ordner *PlugIn.Example* benannt.

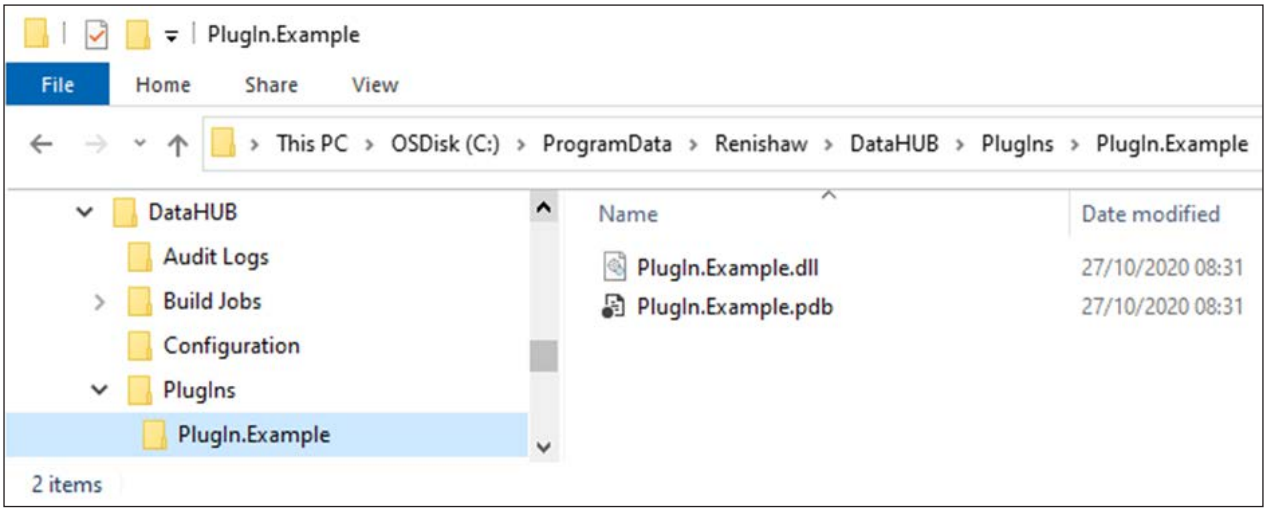


Abbildung 10 Eingefügtes Plug-in

6.3.4.1 Plug-in registrieren

Zum Registrieren des neuen Plug-ins muss der DataHUB Service neu gestartet werden. Verwenden Sie dazu die *Services*-Anwendung. Klicken Sie mit der rechten Maustaste auf *DataHUB Service* und wählen Sie *Restart* (Neustart):

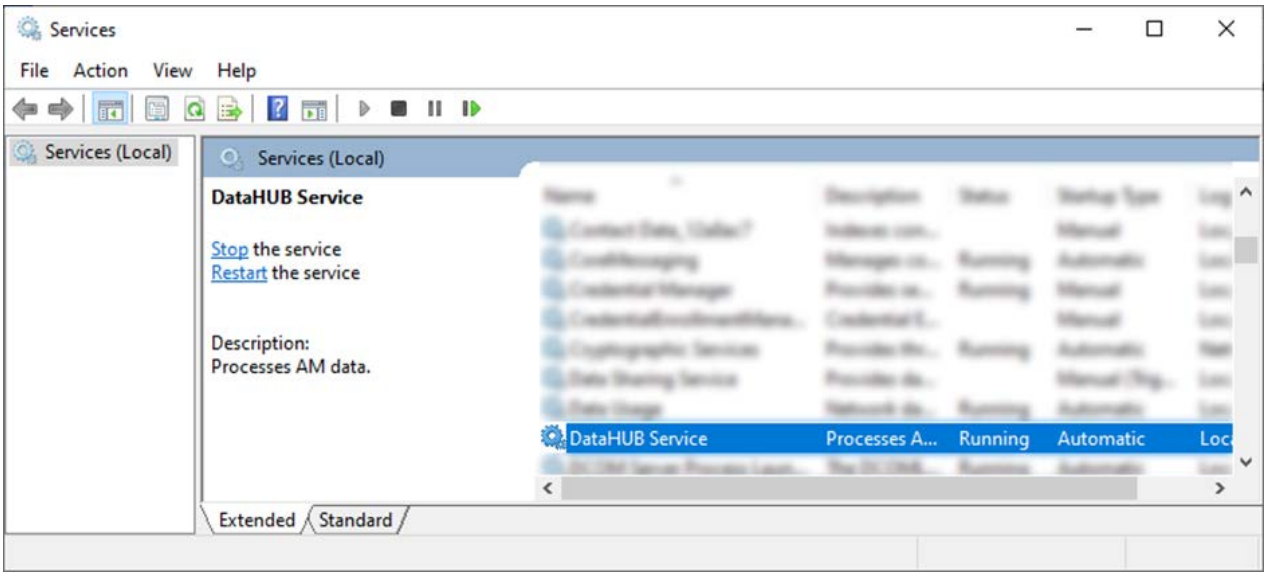


Abbildung 11 Windows Service-Anwendung

Nach einigen Sekunden wird für den DataHUB Service der Status *Running* (Läuft) gemeldet.

6.3.4.2 Registrierung überprüfen

Beim Ausführen von DataHUB wird eine Protokolldatei erzeugt, in der wichtige Ereignisse wie Baubeginn, Baufehler und natürlich die Plugin-Überprüfung festgehalten werden. Suchen Sie im Datei-Explorer nach dem Protokollordner <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>.

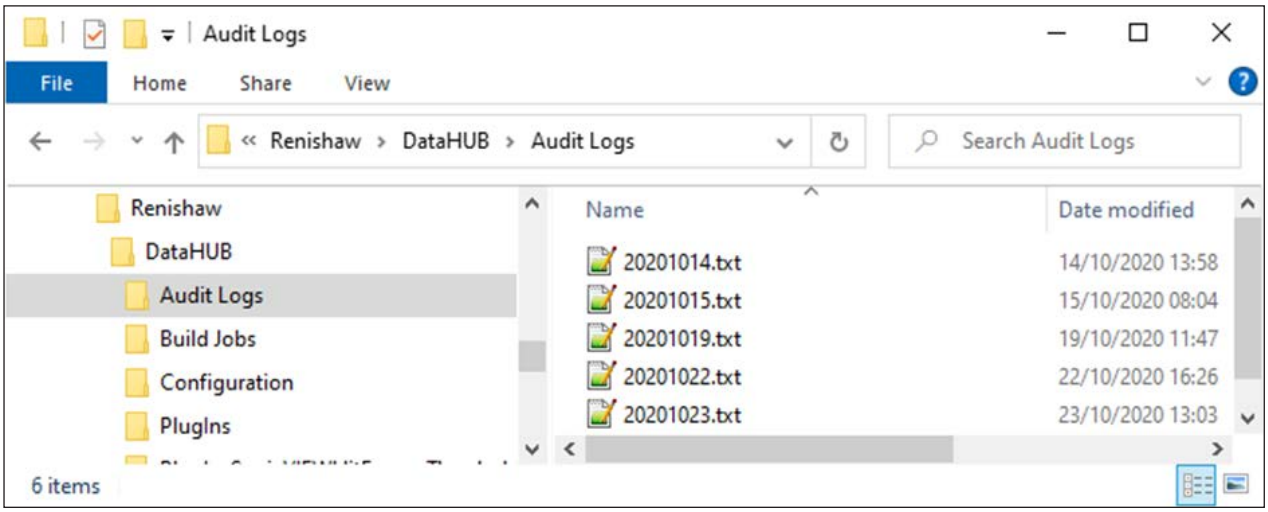


Abbildung 12 DataHUB Protokollordner

Öffnen Sie die jüngste Protokolldatei – sie sollte das aktuelle Datum aufweisen – blättern Sie bis zum Ende und suchen Sie nach Zeilen mit folgenden Angaben, um gültige Plug-ins ausfindig zu machen:

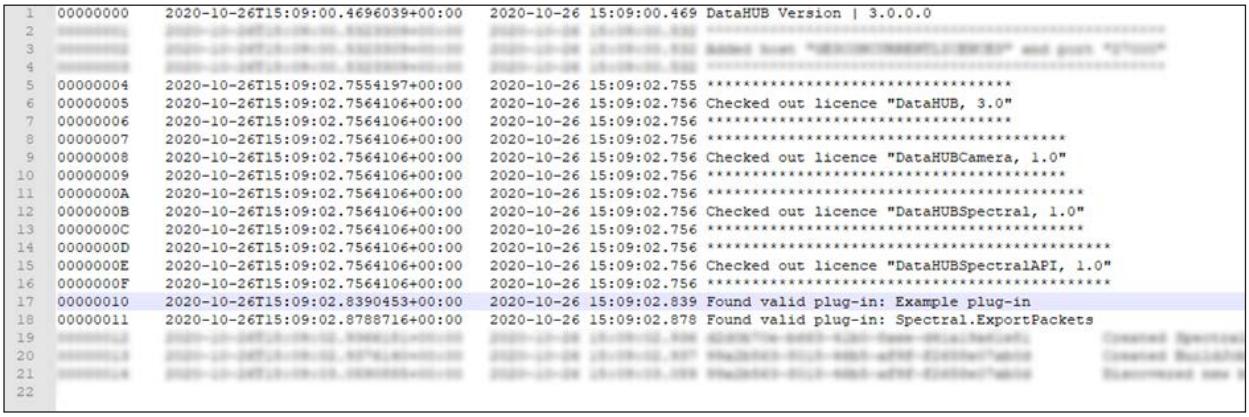


Abbildung 13 Von DataHUB gefundene gültige Plug-ins

6.3.5 Zeitlimit (Timeout)

Da DataHUB nicht garantieren kann, dass auf jeden Aufruf einer Plug-in-Instanz eine zeitnahe Rückmeldung erfolgt, wird eine Timeout-Regel angewendet. Wenn ein Plug-in-Aufruf nicht innerhalb einer bestimmten Zeit abgeschlossen wird, gilt er als fehlerhaft und wird abgebrochen. Die Standard-Timeouts sind:

- Für *Initialise*: 20 Sekunden
- Für *ProcessPackets*: 200 Sekunden
- Für *Shutdown*: 20 Sekunden

Beim Debuggen werden diese Timeouts häufig überschritten und es kann vorkommen, dass DataHUB das Plug-in vorzeitig beendet. Um mehr Zeit für das Debuggen zu ermöglichen, können die Timeout-Limits in der Datei <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> mit einem geeigneten XML- oder Texteditor erhöht werden.

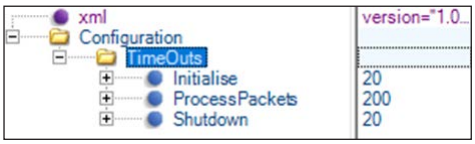


Abbildung 14 Standard-Timeouts für Plug-ins

HINWEIS: Damit die Timeout-Änderungen wirksam werden, muss der DataHUB Service neu gestartet werden.

6.3.6 Plug-in debuggen

Nachdem das Plug-in erfolgreich eingefügt wurde, ist es nun möglich, seine Ausführung zu debuggen. Das Plug-in selbst ist nicht direkt ausführbar, aber wenn Sie den Visual Studio Debugger an den DataHUB Service anfügen und einen Bauauftrag starten, ruft der Service die Plug-in-Implementierung auf, die an den im Code vorgesehenen Haltepunkten gestoppt wird.

HINWEIS: Zum Debuggen über den DataHUB Service benötigt Visual Studio möglicherweise Administratorrechte. Klicken Sie dazu mit der rechten Maustaste auf die Visual Studio-Programmverknüpfung und wählen Sie **Run as administrator** (Als Administrator ausführen).

Fügen Sie in Visual Studio zu Beginn der Methoden *Initialise* und *ProcessPackets* Haltepunkte ein:

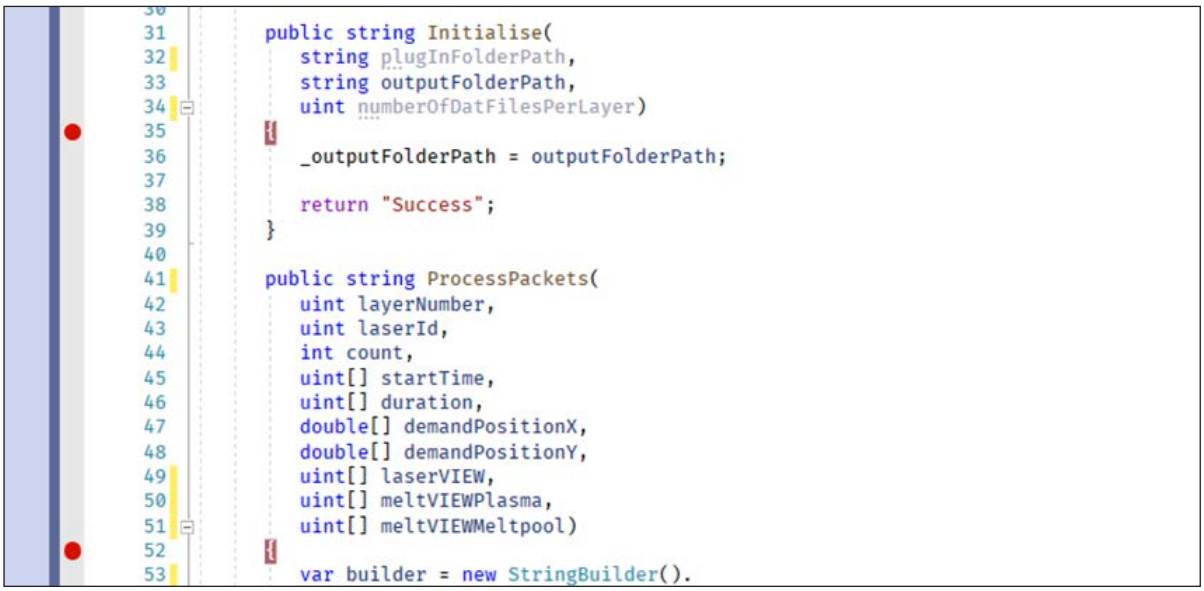


Abbildung 15 Haltepunkte in den Methoden *Initialise* und *ProcessPackets*

Öffnen Sie das Menü **Debug** (Debuggen) und wählen Sie **Attach to process…** (An Prozess anfügen):

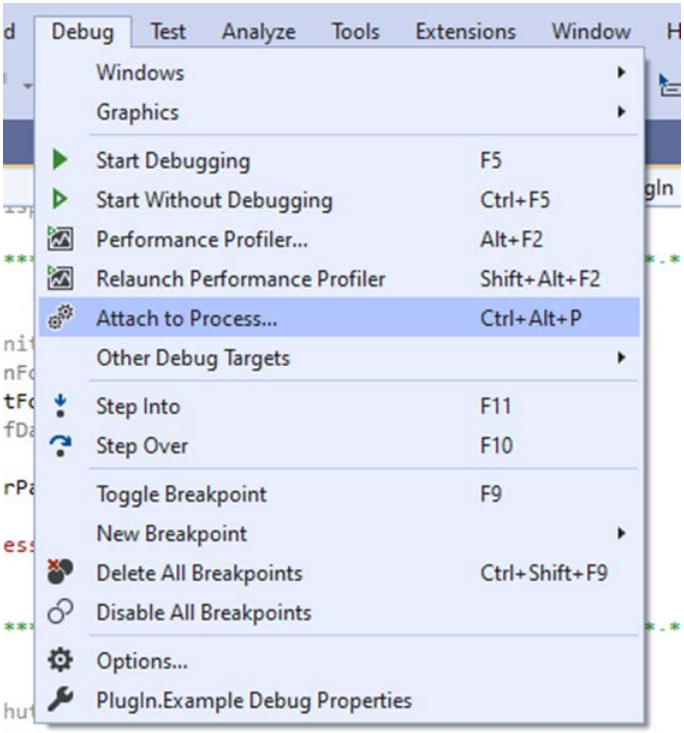


Abbildung 16 Zum Debuggen an einen Prozess anfügen

Suchen Sie in der Liste der laufenden Prozesse die Datei *DataHUB Service.exe*. Möglicherweise müssen Sie die Option **Show processes from all users** (Prozesse aller Benutzer anzeigen) aktivieren.

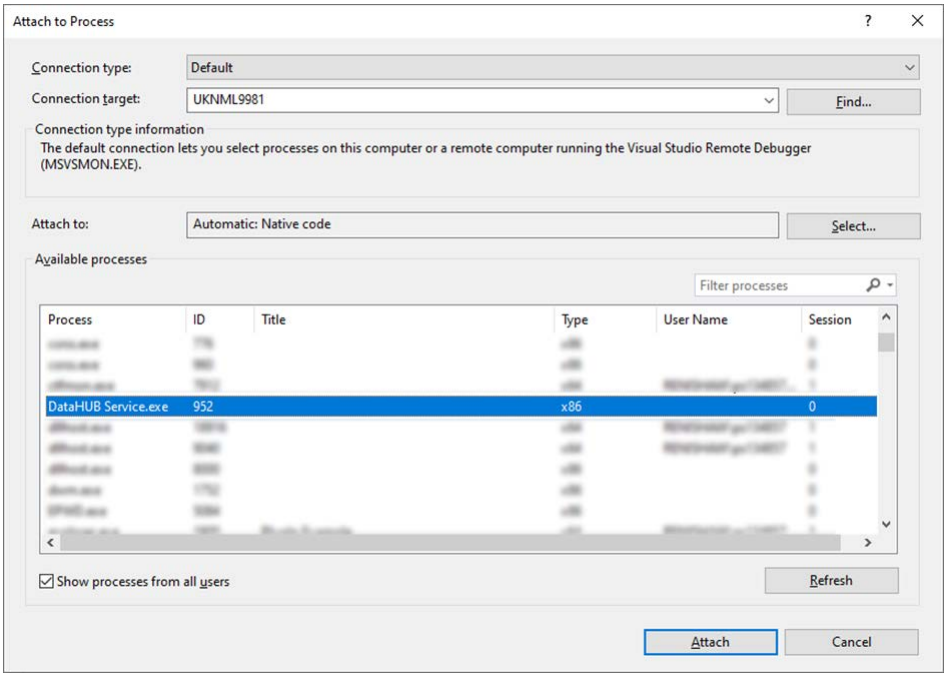


Abbildung 17 DataHUB Service Prozess

Drücken Sie die Schaltfläche **Attach** (Anfügen). Nun beginnt Visual Studio beginnt eine Debugging-Session für den DataHUB Service.

Damit DataHUB das Plug-in aufruft und somit die Haltepunkte getroffen werden können, muss ein Bau gestartet werden. Dies kann über den DataHUB Generator erfolgen. Wenn Sie bereits einige Bauvorgänge generiert haben, können Sie aber auch einfach einen Bauordner in <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs> duplizieren.

DataHUB erkennt den neuen Bau automatisch und beginnt mit der Verarbeitung der Daten. Eine der ersten Aufgaben des Dienstes besteht darin, eine Instanz jedes registrierten Plug-ins zu erstellen (durch Aufrufen des parameterlosen Konstruktors). Wenn DataHUB dann die Instanz der Initialisierungsmethode *Initialise* aufruft, ist der erste Haltepunkt erreicht. Als Methodenparameter werden die Werte des betreffenden Baus eingesetzt, zum Beispiel:

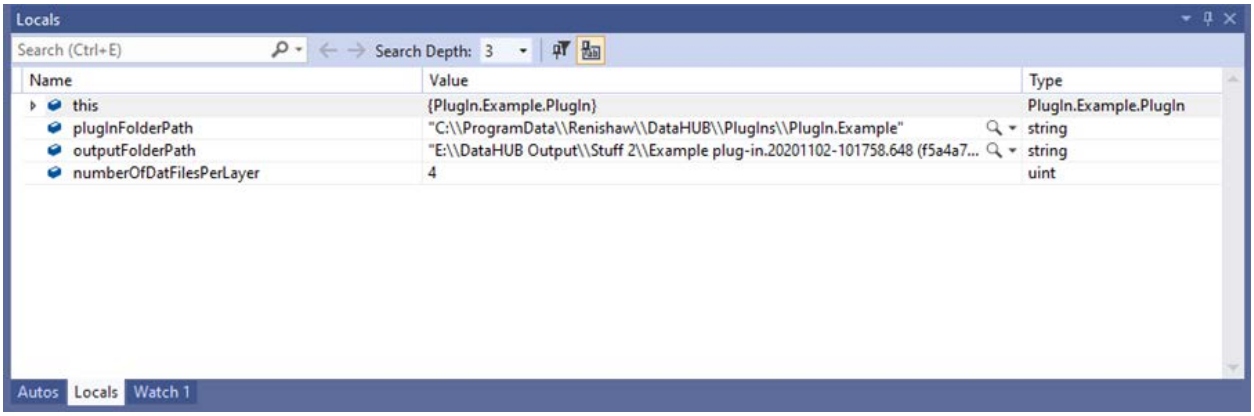


Abbildung 18 Parameterwerte der *Initialise*-Methode

Von diesem Punkt an werden vorhandene Daten (bei einem laufenden Bau auch neue Daten, sobald sie eintreffen) verarbeitet, und die *ProcessPackets*-Methode der Instanz wird mit Parameterwerten für die LaserVIEW- und MeltVIEW-Daten aufgerufen, die einer Schicht zugeordnet sind:

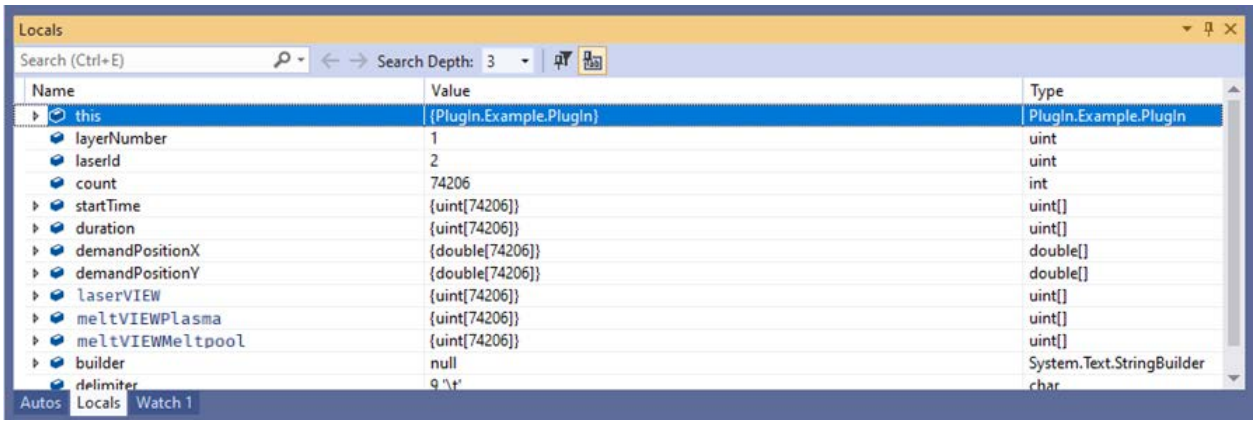


Abbildung 19 Parameterwerte der *ProcessPacket*-Methode

Das eigenständige Hinzufügen und Überprüfen von Haltepunkten in der *Shutdown*-Methode bleibt dem Leser überlassen.

6.3.7 Anwendung des Plug-ins in der Produktion

Um die Anwendung des Plug-ins auf einem Produktions-DCPC vorzubereiten, muss ein Release-Build kompiliert, lokal eingesetzt und getestet werden. Sobald die Funktionalität des Plug-ins validiert und verifiziert wurde, kann es auf Produktions-DCPCs eingesetzt werden.

Die Bereitstellung auf einem Produktions-DCPC verläuft ähnlich wie die Bereitstellung auf einem Entwicklungs-PC. Der Ordner *Release* sollte in den Unterordner *PlugIns* im DataHUB Datenordner (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) auf dem DCPC kopiert und passend zum dem Namen des Plug-ins umbenannt werden. Der DataHUB Service muss dann neu gestartet werden, um das neue Plug-in zu registrieren. Verwenden Sie hierzu die *Services* Anwendung. Die erfolgreiche oder fehlgeschlagene Registrierung des neuen Plug-ins wird im Audit-Log des Tages protokolliert.

HINWEIS: DataHUB ist so konzipiert, dass alle laufenden Datenverarbeitungsaufgaben nach einem Neustart wieder aufgenommen werden. Es ist also nicht notwendig, vor dem Neustart zu warten, bis alle laufenden Aufgaben abgeschlossen sind. Allerdings wird das neu registrierte Plug-in nur für neue Datenverarbeitungsaufgaben angewendet.

6.3.8 Diagnose von Plug-in-Fehlern in der Produktion

In der Produktionsumgebung zeigt DataHUB Monitor (neben anderen Verarbeitungsaufgaben) den Verarbeitungsstatus für das Plug-in an:

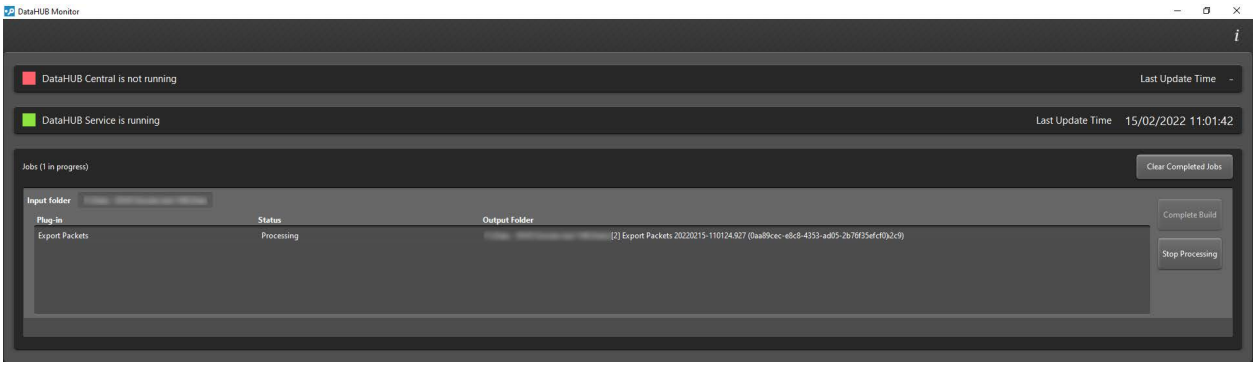


Abbildung 20 Laufender Bau mit Plug-ins

Lautet der Status eines Plug-ins *Error* (Fehler), so wird dieser Fehler im Audit-Protokoll erfasst. Dies erfolgt entweder in Form von Einträgen, die von DataHUB selbst hinzugefügt wurden (z. B. fehlgeschlagenes Laden der Plug-in-Assembly aufgrund fehlender Abhängigkeiten), oder durch das Plug-in über die Ausgabewerte der Plug-in-Methoden *Initialise*, *ProcessPackets* oder *Shutdown*.

6.3.9 Problemlösung

Die folgende Tabelle enthält häufige Probleme, deren typische Audit-Protokolleinträge und mögliche Abhilfemaßnahmen:

Audit-Protokolleintrag	Problem	Abhilfe
The assembly does not contain a class named PlugIn	Die Plug-in-Klasse wurde nicht gefunden.	Plug-in-Klasse umbenennen, erneut kompilieren und bereitstellen.
The PlugIn class does not contain a method named xyz	Die Plug-in-Klasse implementiert keine der API-Methoden.	Überprüfen Sie die Klasse auf fehlende oder fehlerhafte Methoden.
The xyz method does not have the expected signature …	Die Plug-in-Klasse implementiert eine der API Methoden nicht korrekt.	Prüfen Sie die Signatur der Methode und stellen Sie sicher, dass alle Parameter mit der API-Definition übereinstimmen.
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.	Die Plug-in-Klasse gibt keine gültige Zeichenfolge für die API-Version zurück.	Stellen Sie sicher, dass die APIVersion-Methode den Wert „1.0“ zurückgibt.
Failed to create plug-in output folder <path>	Der Ordner für eine bestimmte Instanz eines Plug-ins konnte nicht erstellt werden.	Stellen Sie sicher, dass die Zugriffsrechte für das Lesen/Schreiben/Ändern/Erstellen auf <Pfad> vorhanden sind.
Initialisation result: timed out Process result: timed out Shutdown result: timed out	Die Rückgabe nach dem Aufruf einer dieser API-Methoden hat zu lange gedauert. Das Plug-in ist vermutlich fehlgeschlagen.	Ziehen Sie in Erwägung, den Arbeitsaufwand in der Methode zu verringern oder das Timeout in der Datei PlugInsKonfiguration.xml zu verlängern.
Exception:	Ein unerwarteter Fehler ist aufgetreten.	Untersuchen Sie die protokollierten Ausnahme-meldungen, um die Fehlerquelle zu ermitteln.

6.3.10 Integration mit Renishaw Central

Die Analyseergebnisse (Zeitseriendatenpunkte, Meldungen usw.) eines Plug-ins können an einen Renishaw Central Server weitergeleitet werden. Im Dokument *Plug-in API V2.0 und Tutorials* (Renishaw Artikel-Nr. H-5800-6784) finden Sie eine ausführliche Erklärung, wie der Inhalt des von den Plug-in-Methoden *Initialise* und *ProcessPackets* zurückgegebenen Strings zu strukturieren ist, um diese Integration zu erreichen, sowie Einzelheiten zu Hilfsfunktionen in der Datei *PlugIns.API.v2.dll*, die mit der API V1.0 kompatibel sind.

6.3.11 Die Plug-in API V1.0

6.3.11.1 Assembly benennen

Der Dateiname der .NET Assembly muss mit „PlugIn.“ beginnen („PlugIn“ gefolgt von einem Punkt) und die Dateinamenerweiterung „dll“ haben.

6.3.11.2 Plug-in-Klasse benennen

Die Plug-in-Assembly muss eine öffentliche (public) Klasse namens `PlugIn` definieren.

6.3.11.3 Methoden für Plug-in-Klassen

Die Plug-in-Klasse muss die folgenden öffentlichen (public) Methoden implementieren:

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

6.3.11.4 Der parameterlose Konstruktor

Der Typ muss einen parameterlosen Konstruktor haben. Wenn keine Konstruktoren mit Parametern definiert sind, macht C# diesen automatisch verfügbar, d. h., es ist nicht notwendig, ihn explizit zu definieren.

HINWEIS: Ein Plug-in sollte innerhalb seines Konstruktors keine Ressourcen akquirieren.

Bei einer Plug-in-Instanz mit Konstruktor ist nicht gewährleistet, dass ihre *Shutdown*-Methode aufgerufen wird. Solche Ressourcen werden folglich möglicherweise nicht rechtzeitig bereinigt. Ressourcen sollten über die Methode *Initialise* akquiriert werden.

6.3.11.5 Die `APIVersion`-Methode

Diese Methode gibt vor, gegen welche Version der API dieses Plug-in validiert werden soll, sowie das Format der Prozessüberwachungsdaten, die es verarbeiten wird.

Parameter:

Keine.

Rückgabe: `string`

Ein Plug-in, das API V1.0 implementiert, muss das Literal 1.0 zurückgeben.

6.3.11.6 Die `DisplayName`-Methode

Diese Methode gibt den Namen zurück, der als Plug-in-Name im DataHUB Monitor, beim Erstellen des Ausgabeordners für das Plug-in und beim Erfassen von Plug-in-Ereignissen angezeigt wird. Dieser Name muss zwar nicht zwingend eindeutig sein, dies erleichtert jedoch das Identifizieren des Plug-ins, wenn z. B. mehrere Versionen auf einem DCPC installiert sind.

Parameter:

Keine.

Rückgabe: `string`

Das in verschiedenen Teilen der Benutzeroberfläche von DataHUB, zur Protokollierung usw. verwendete Literal.

6.3.11.7 Die `Initialise`-Methode

Diese Methode wird von DataHUB zu Beginn eines Baus aufgerufen, bevor irgendwelche Schichtdaten an das Plug-in gesendet werden, und erhält daher *bauinvariante* Parameterwerte. Das Plug-in kann diese Methode verwenden, um Ressourcen zuzuweisen, die während der Lebensdauer der Instanz zur Berechnung von Baukonstanten usw. benötigt werden.

Parameter 1: `string`

Der Pfad zum Ordner mit der Plug-in-Assembly, z. B. <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugIn.Example>

Parameter 2: `string`

Der Pfad eines Ordners, in den dieses Plug-in Ausgabedateien schreiben darf. Beachten Sie, dass dieser Ordner für jedes Plug-in und für jeden Bau eindeutig ist und mit dem Anzeigenamen des Plug-ins (durch den von der `DisplayName`-Methode zurückgegebenen Wert definiert), einem Zeitstempel und einer GUID benannt wird. Der Ordnername ändert sich jedes Mal, wenn das Plug-in für einen Bau ausgeführt wird, folgt aber immer dieser Struktur:

<Bau-Ausgabeordner>\[1.0] <Plug-in Anzeigename>.yyyyMMdd-HHmmss.fff (GUID)

Parameter 3: `uint`

Dies ist ein Wert von 1 bis 4, der durch die Hardwarekonfiguration des Rechners, der die Daten erstellt, bestimmt wird.

Rückgabe: `string`

Das Plug-in sollte einen String zurückgeben, der entweder:

- das Wort „Success“ (Erfolg) enthält, wenn die Methode korrekt abgeschlossen wurde, oder
- beliebige Angaben zur Fehlerursache enthält oder
- eine JSON-Zeichenkette enthält, die der Renishaw Central Return API entspricht.

Wenn DataHUB eine nicht konforme Zeichenkette empfängt, wird das Plug-in beendet und der Inhalt der zurückgegebenen Zeichenkette zur späteren Diagnose in das Audit-Protokoll aufgenommen.

6.3.11.8 Die `ProcessPackets`-Methode

DataHUB ruft diese Methode auf, nachdem alle Daten für eine Schicht vom DCPC empfangen wurden. Die Parameter für diese Methode sind:

Parameter 1: `uint`

Die Nummer der Bauschicht, auf die sich die Daten beziehen. Die erste Schicht ist layer 1 (Schicht 1).

Parameter 2: `uint`

Die Identifikationsnummer (1, 2, 3 oder 4) des Lasers, für den die Daten erfasst wurden.

Parameter 3: `int`

Die Anzahl der Elemente in den nachfolgenden Arrays.

Parameter 4: `uint[ ]`

Die Startzeit jedes Pakets in µs seit Beginn der Schicht.

Parameter 5: `uint[ ]`

Die Dauer der einzelnen Pakete in µs.

Parameter 6: `double[ ]`

Die angeforderte X-Position jedes Pakets auf dem Pulverbett in mm.

Parameter 7: `double[ ]`

Die angeforderte Y-Position jedes Pakets auf dem Pulverbett in mm.

Parameter 8: `uint[ ]`

Der gemittelte Wert aller Samples, die der LaserVIEW-Sensor während der Dauer eines jeden Pakets erfasst hat.

Parameter 9: `uint[ ]`

Der gemittelte Wert aller Samples, die der MeltVIEW Plasma-Sensor während der Dauer eines jeden Pakets erfasst hat.

Parameter 10: `uint[ ]`

Der gemittelte Wert aller Samples, die der MeltVIEW Meltpool-Sensor während der Dauer eines jeden Pakets erfasst hat.

Rückgabe: `string`

Das Plug-in sollte entweder:

- einen String zurückgeben, der das Wort „Success“ (Erfolg) enthält, wenn die Methode korrekt abgeschlossen wurde, oder
- einen String mit möglichst detaillierten Angaben zur Fehlerursache zurückgeben oder
- eine JSON-Zeichenkette zurückgeben, die der Renishaw Central Return API entspricht.

6.3.11.9 Die `Shutdown`-Methode

DataHUB ruft diese Methode auf, nachdem alle Daten für einen Bau an das Plug-in gesendet wurden oder wenn das Aufrufen der Methode *Initialise* fehlgeschlagen ist. Mit dieser Methode soll das Plug-in sämtliche abschließenden Verarbeitungen an den Baudaten vornehmen, alle eventuell übernommenen Ressourcen freigeben usw. Die Methode hat keine Parameter.

Parameter:

Keine.

Rückgabe: `string`

Das Plug-in sollte einen String zurückgeben, der entweder:

- das Wort „Success“ (Erfolg) enthält, wenn die Methode korrekt abgeschlossen wurde, oder
- möglichst detaillierte Angaben zur Fehlerursache enthält.

Wenn DataHUB einen nicht konformen String empfängt, wird der Inhalt der zurückgegebenen Zeichenkette zur späteren Diagnose in das Audit-Protokoll aufgenommen.

6.4 API V2 Plug-in

Dieser Abschnitt enthält die Anleitung zum Erstellen, Testen und Bereitstellen einer Plug-in-Softwarekomponente V2.0 für DataHUB. Der Plug-in-Entwicklungszyklus beginnt mit der Installation von DataHUB auf dem Entwicklungs-PC. Danach wird in Visual Studio eine .NET-Assembly mit dem Plug-in-Code erstellt. Testen und debuggen Sie das Plug-in anschließend mit der entsprechenden Visual Studio-Debugging-Konfiguration, um dessen korrekte Funktionsweise zu verifizieren, bevor es auf dem Rechner, der in einem AM-Produktionssystem zur Datenerfassung dient (Data Collector PC bzw. DCPC) bereitgestellt wird.

In diesem Abschnitt wird die Struktur eines *Token-Streams* für den Bau definiert. Ein Code-Beispiel zeigt dann, wie man ein Plug-in zur Verarbeitung des Token-Streams für die API V2.0 schreibt. Das zweite Codebeispiel demonstriert das Verwenden von *Filtern* zur Verarbeitung eines Token-Streams. Anschließend wird die Implementierung des *ExportPackets* Plug-ins für den Export der Pakete (mit der DataHUB Software gelieferten) neu erstellt. Die letzten Code-Beispiele zeigen, wie die Ergebnisse eines Plug-ins an Renishaw Central gesendet werden können, um sie gemeinsam mit den Daten darzustellen, die von anderen Renishaw Systemen für die additive Fertigung erfasst wurden.

Dieser Abschnitt schließt mit einer Definition der API V2.0.

6.4.1 Voraussetzungen und vorbereitende Schritte

Lesen Sie in Verbindung mit diesem Handbuch auch die folgenden Dokumente:

- *InfiniAM® und DataHUB Software* Installationshandbuch (Renishaw Art. Nr. H-5800-6844)
- *DataHUB* Benutzerhandbuch (Renishaw Art. Nr. H-5800-6852)
- Benutzerhandbuch für Renishaw Licence Manager v2.x

Bevor Sie fortfahren, sollten Sie die folgenden vorbereitenden Schritte auf dem PC durchführen, der für die Plug-in-Entwicklung verwendet werden soll:

- Vergewissern Sie sich, dass der PC mit einem Grafikprozessor ausgestattet ist, der zumindest die DataHUB Mindestanforderungen erfüllt (siehe die Hardware-Spezifikationen für den Datenerfassungs-PC im Installationshandbuch für *InfiniAM und DataHUB Software*).
- Vergewissern Sie sich, dass die aktuellen Treiber für die GPU installiert sind.
- Prüfen Sie, dass der Lizenzserver über genügend geeignete Lizenzen für DataHUB und Spectral API verfügt.
- Überprüfen Sie den Netzwerkzugriff auf den Lizenzserver.
- Stellen Sie sicher, dass eine Version von Microsoft Visual Studio installiert ist, die das .NET Framework 4.7.2 unterstützt.

6.4.2 Token-Streams

6.4.2.1 Aufbau des Token-Streams

Der DataHUB Service wandelt die Prozessüberwachungsdaten eines Baus in einen Token-Stream um, der von V2.0-Plug-ins verwendet werden kann. Die Reihenfolge und der Token-Typ in einem Token-Stream bilden eine pseudohierarchische Beschreibung der Prozessüberwachungsdaten.

Die Tokens in einem Stream fallen in zwei Kategorien: gepaarte Start- und Endtokens oder Tokens, die durch gepaarte Start- und End-Tokens begrenzt sind.

6.4.2.2 Token-Stream für einen Laser und eine Schicht

Der folgende Token-Stream stellt die Prozessüberwachungsdaten für einen Laser und eine Schicht eines Baus dar:

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

Beachten Sie, dass das Token-Paar `StartOfLayerLaserData` und `EndOfLayerLaserData` die laserspezifischen Tokens konzeptionell eingrenzt.

Das Token `DataSourceInformation` enthält Werte aus den Datenquellen der Prozessüberwachung (d. h. DAT-Dateien) mit den Sample-Daten.

Jedes `Sample`-Token enthält die Prozessüberwachungsdaten, die von der LaserVIEW- und MeltVIEW-Hardware für ein einzelnes 100-KHz-Sample erfasst wurden.

6.4.2.3 Token-Stream für eine Schicht

Der Satz aller Prozessüberwachungsdaten für eine Schicht wird zu einem Token-Stream zusammengefasst, wobei die oben beschriebene Struktur pro Laser für jeden Laser der Maschine einmal wiederholt wird:

```
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)
```

Auch hier sind die schichtspezifischen Tokens wiederum vom Token-Paar `StartOfLayerData` und `EndOfLayerData` umgeben.

6.4.2.4 Token-Stream für einen Bau

Die gesamten Daten für einen Bau bilden einen Token-Stream, der die oben beschriebene Struktur für jede Schicht wiederholt:

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

6.4.2.5 Split-Tokens

Ein kleines Detail wurde in der soeben beschriebenen Token-Stream-Struktur nicht erwähnt: `Split`-Tokens. Split-Tokens werden dem Token-Stream hinzugefügt, um Punkte zu markieren, an denen sich die Daten ändern. Sie werden als generisches Signal zusätzlich zu den spezifischeren `StartOf/EndOf`… Tokens hinzugefügt. Bei der Verarbeitung eines Token-Streams ist es oft einfacher, auf dieses generische Token zu reagieren als auf spezifischere Marker.

6.4.2.6 Vollständige Definition des Token-Streams für einen Bau

Die vollständige Definition eines Token-Streams für einen Bau lautet:

```
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 Token-Typ hinzufügen

Die API definiert eine grundlegende `Token`-Klasse, von der alle Tokens abgeleitet werden. Um einen neuen Token-Typ in den Token-Stream aufzunehmen, leiten Sie ihn von dieser Klasse ab:

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

Einen vollständigen Katalog der API-Tokens finden Sie in Abschnitt 6.4, „API V2 Plug-in“.

6.4.3 DataHUB für die Plug-in-Entwicklung installieren

Der DataHUB Service muss auf dem Entwicklungs-PC installiert werden. Im Laufe der Installation können Sie wählen, welche Lizenzen genutzt werden sollen. Vergewissern Sie sich, dass die Option **LaserVIEW/ MeltVIEW Pluggability** (LaserVIEW/MeltVIEW Plug-in-Kompatibilität) aktiviert ist.

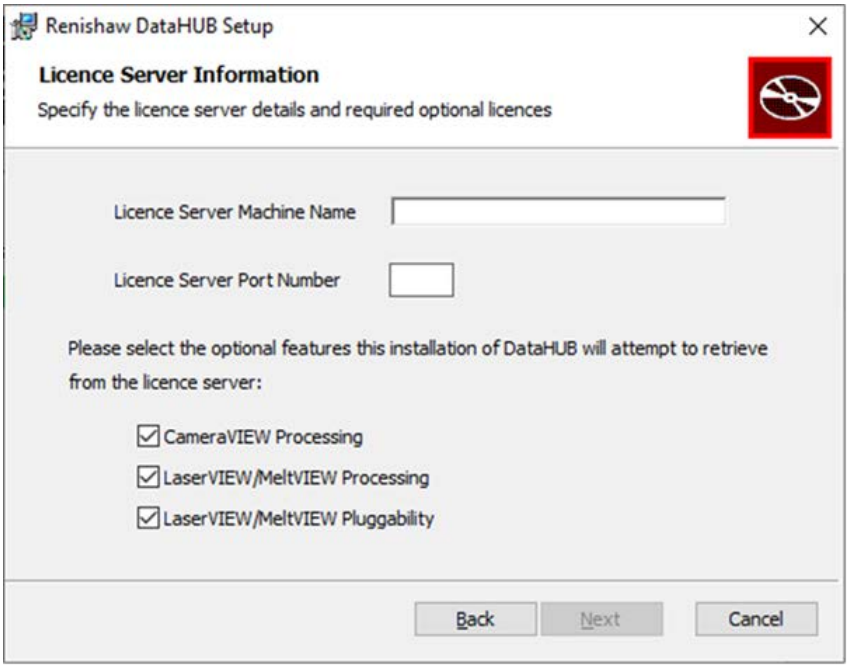


Abbildung 21 DataHUB Lizenzen

6.4.4 Erstellen eines Plug-ins in Visual Studio

Die folgenden Abschnitte erläutern das Erstellen einer Plug-in-Lösung in Visual Studio.

Plug-ins werden in C# geschrieben und definieren eine öffentliche Klasse (public class), die den spezifischen Satz öffentlicher Methoden implementiert, anhand derer DataHUB beim Verarbeiten der Prozessüberwachungsdaten eines Baus Instanzen des Plug-ins identifiziert, validiert und verwendet.

HINWEIS: Im unten gezeigten Ablauf wurde Microsoft Visual Studio 2019 verwendet, die Vorgehensweise ist aber in anderen Microsoft Visual Studio Versionen weitgehend ähnlich.

6.4.4.1 Lösung erstellen

Starten Sie Visual Studio und wählen Sie *Create a new project* (Neues Projekt erstellen). Wählen Sie unter den verfügbaren Optionen *Class Library (.NET Framework)* (Klassenbibliothek (.NET Framework)):

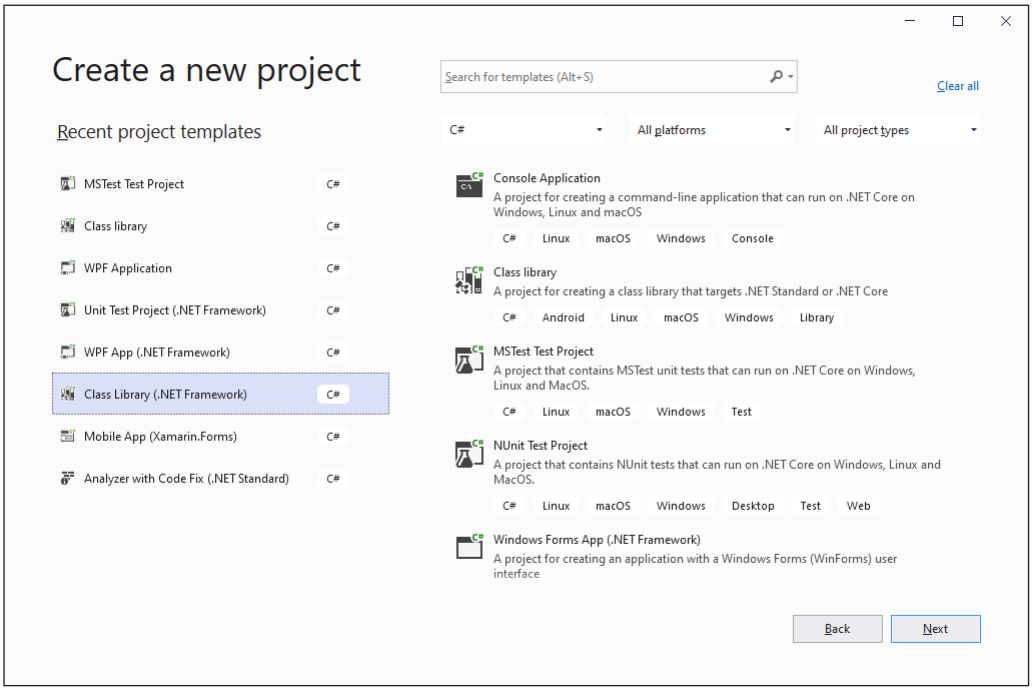


Abbildung 22 Lösung erstellen

Nun müssen Sie das Projekt konfigurieren:

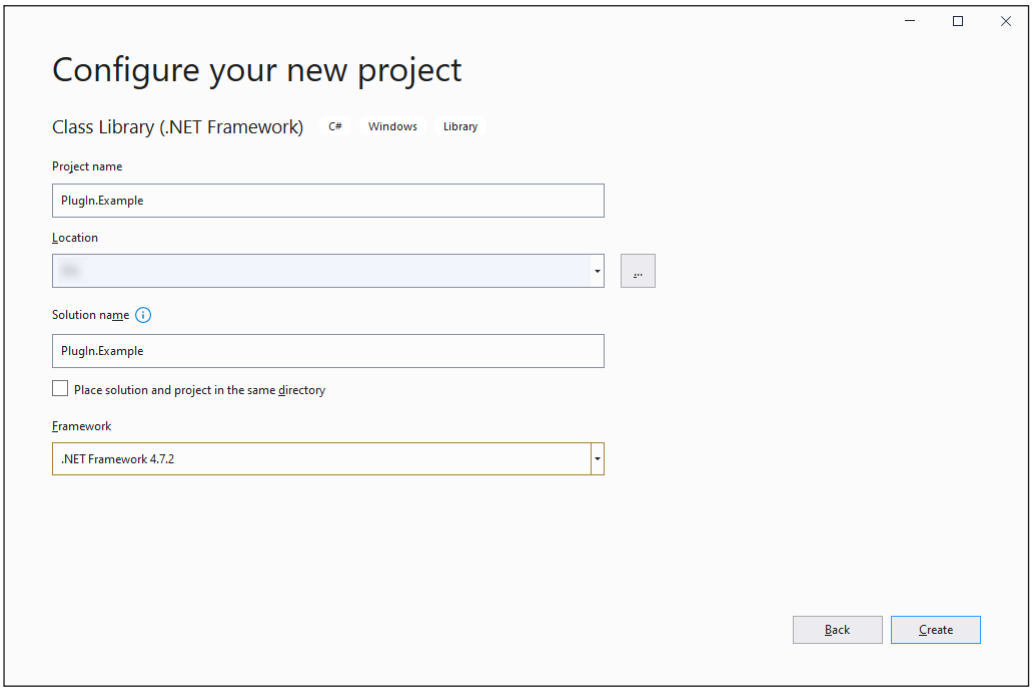


Abbildung 23 Projekt konfigurieren

DataHUB erfordert, dass eine Plug-in-Assembly mit **PlugIn.** beginnt (das Wort „PlugIn“ gefolgt von einem Punkt). Wählen Sie unter *Location* (Speicherort), wo das Projekt gespeichert werden soll, und erzeugen Sie das Plug-in.

HINWEIS: DataHUB unterstützt Plug-ins, die auf x86/x64/Any CPU ausgerichtet sind. Als Ziel wird Framework 4.7.2 oder höher empfohlen.

6.4.4.2 Einer Plug-in API-Assembly einen Verweis hinzufügen

Um ein Plug-in für die API V2.0 zu entwickeln, ist ein Verweis auf die API-Assembly des Plug-ins erforderlich. Klicken Sie mit der rechten Maustaste auf den Eintrag **References** (Verweise) im Projekt und wählen Sie **Add Reference...** (Verweis hinzufügen).

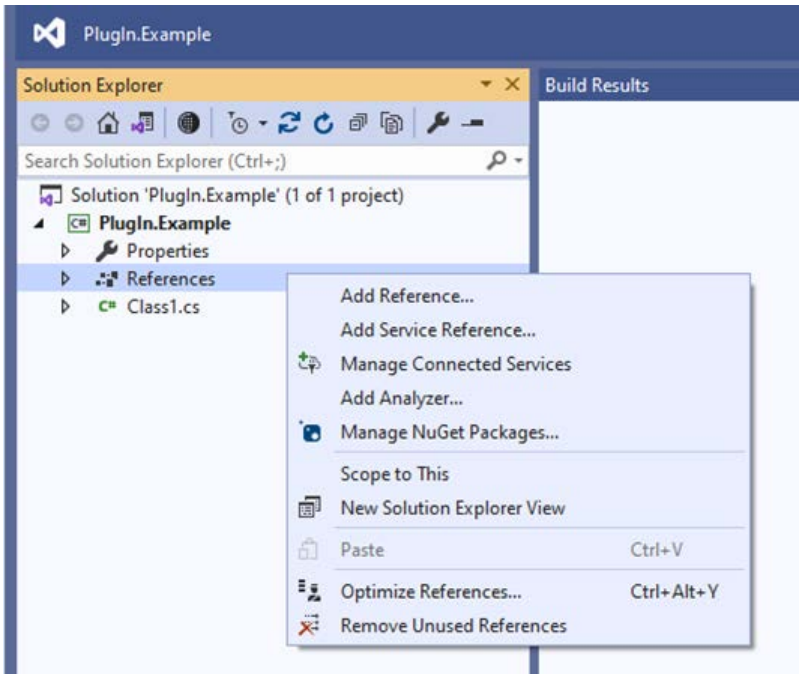


Abbildung 24 Verweis hinzufügen

Gehen Sie im Verweis-Manager auf „Browse…", and locate the "PlugIns.API.v2.dll" in the DataHUB installation folder (i.e., <$PROGRAMFILES$\Renishaw\DataHUB>):

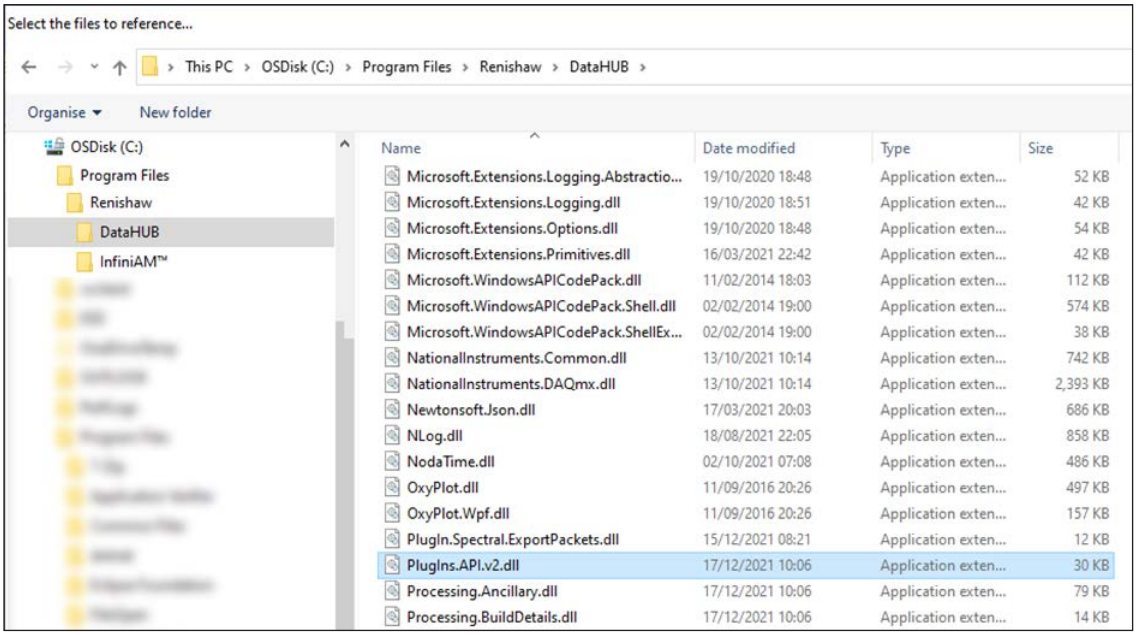


Abbildung 25 Die API-Assembly

Die Assembly wird dann unter den Abhängigkeiten des Plug-in-Projekts gelistet:

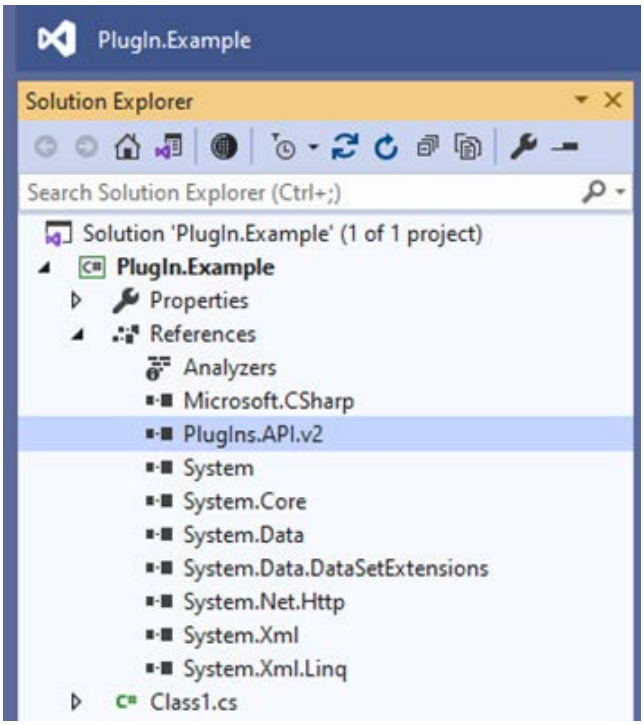


Abbildung 26 Plug-in API-Assembly wird als Abhängigkeit gelistet

6.4.4.3 Plug-in-Typ benennen

Die Standard-Quelldatei Class1.cs enthält die Definition von Class1, die wir als Ausgangspunkt verwenden.

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

Sobald DataHUB eine potenzielle Plugin-Assembly entdeckt hat, sucht es nach einem `PlugIn` des Typs „Public“. Benennen Sie folglich `Class1` zu `PlugIn` um:

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

6.4.4.4 API hinzufügen

DataHUB verlangt, dass potenzielle Plug-in-Typen die folgenden Public-Methoden implementieren:

```
public PlugIn() (wird automatisch erzeugt)
public static string APIVersion()
public static string DisplayName()
public string Initialise(…)
public string Process(…)
public string Shutdown()
```

Fügen Sie der Klasse die folgenden Anweisungen und Methodenimplementierungen hinzu:

```
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();
}
```

Beachten Sie, dass alle API-Methoden einen `string` zurückgeben. Der Inhalt dieser zurückgegebenen Zeichenfolge signalisiert DataHUB das Ergebnis der Methode. Die Rolle des Rückgabewerts wird nachfolgend bei der eingehenden Erläuterung der einzelnen Methoden näher erklärt.

6.4.4.5 Die statische `APIVersion`-Methode

Diese Methode meldet DataHUB die Version der API, die das Plug-in verwendet. Für API-Version 2.0 muss sie das Literal 2 zurückgeben:

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

6.4.4.6 Die statische `DisplayName`-Methode

Diese Methode gibt den Namen zurück, der als Plug-in-Name vom DataHUB Monitor, beim Erstellen des Ausgabeordners für das Plug-in usw. angezeigt wird. Dieser Name muss zwar nicht zwingend eindeutig sein, dies erleichtert jedoch das Identifizieren eines bestimmten Plug-ins, wenn z. B. mehrere Versionen auf einem DCPC installiert sind.

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

6.4.4.7 Die `Initialise`-Methode

DataHUB ruft diese Methode zu Beginn eines Baus auf, bevor irgendwelche Schichtdaten an das Plug-in gesendet werden. Sie liefert dem Plug-in folglich *bauinvariante Daten*. In ihrer Mindestimplementierung gibt diese Methode zurück, dass das Plug-in problemlos initialisiert wurde:

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

6.4.4.8 Die `Process`-Methode

DataHUB ruft diese Methode auf, nachdem alle Daten für eine Schicht empfangen wurden. In ihrer Mindestimplementierung gibt diese Methode zurück, dass das Plug-in die Daten problemlos verarbeitet hat:

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

6.4.4.9 Die `Shutdown`-Methode

DataHUB ruft diese Methode auf, nachdem alle Daten für einen Bau an das Plug-in gesendet wurden oder wenn ein vorheriger Plug-in-Aufruf fehlgeschlagen ist. Auch hier besteht die Mindestimplementierung darin, die problemlose Durchführung zu melden:

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

6.4.4.10 Plug-in testen und debuggen

Erstellen Sie die Lösung mit *Debug*, öffnen Sie einen Dateibrowser und suchen Sie den Ausgabeordner, der eine *.dll*-Datei mit dem Namen des Projekts enthält, zum Beispiel *PlugIn.Example.dll*. (Zusatzdateien wie Debug-Symbole usw. können ebenfalls vorhanden sein).

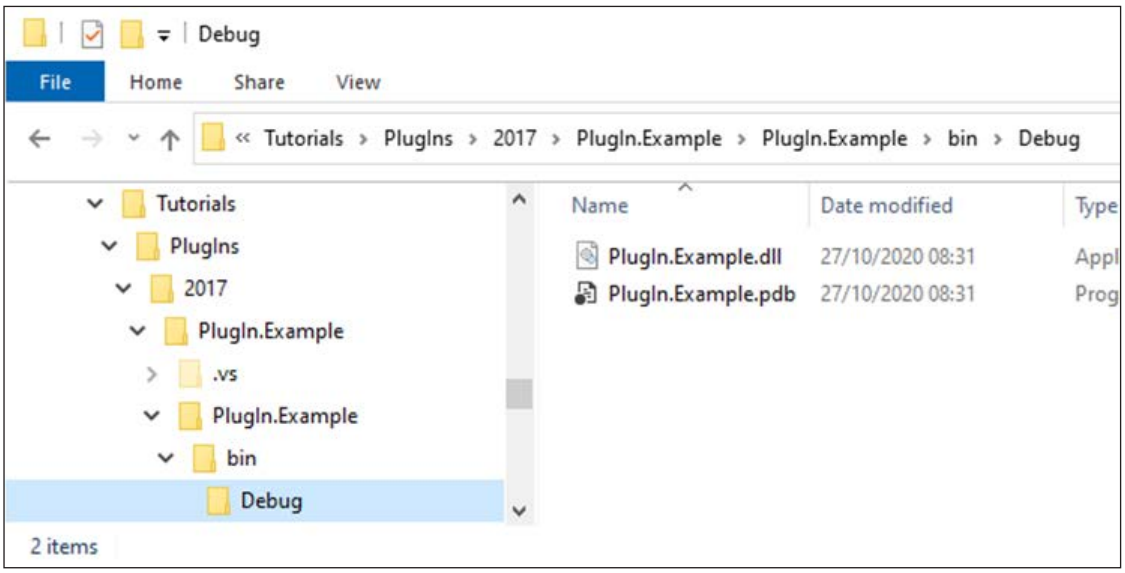


Abbildung 27 Plug-in-Binärdateien

Kopieren Sie diesen Ordner und fügen Sie ihn in den Unterordner „PlugIns“ im Datenordner von DataHUB ein, d. h. <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>.

HINWEIS: Normalerweise befindet sich $PROGRAMDATA$ unter <C:\ProgramData>, der Ordner kann aber auch ausgeblendet sein. Aktivieren Sie in diesem Fall die Option **View, Show/hide, Hidden items** (Ansicht, Erweiterte Einstellungen, Ausgeblendete Dateien, Ordner und Laufwerke anzeigen) im Windows Datei-Explorer.

Um den Inhalt dieses Ordners zu ändern, sind möglicherweise Administratorrechte erforderlich.

Benennen Sie den Ordner *Debug* um, sodass er dem Namen des Plug-ins entspricht. In der nachfolgenden Abbildung wurde der Ordner in *PlugIn.Example* umbenannt.

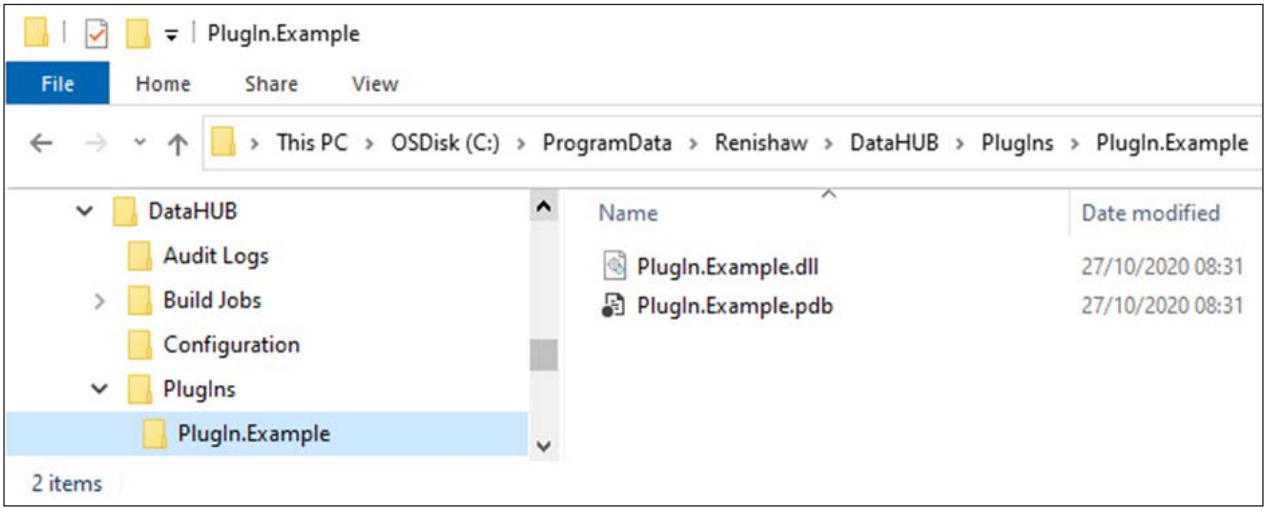


Abbildung 28 Eingefügtes Plug-in

6.4.4.11 Plug-in registrieren

Zum Registrieren des neuen Plug-ins muss der DataHUB Service neu gestartet werden. Verwenden Sie dazu die *Services*-Anwendung. Klicken Sie mit der rechten Maustaste auf *DataHUB Service* und wählen Sie *Restart* (Neustart):

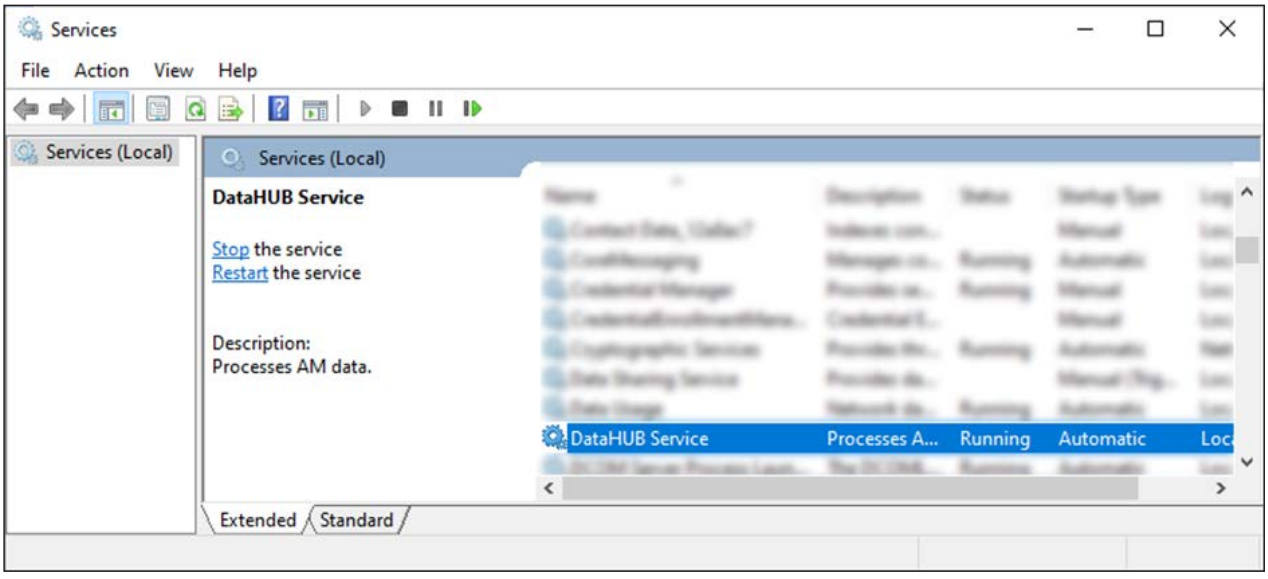


Abbildung 29 Windows Service-Anwendung

Nach einigen Sekunden wird für den DataHUB Service der Status *Running* (Läuft) gemeldet.

6.4.4.12 Registrierung überprüfen

Beim Ausführen von DataHUB wird eine Protokolldatei erzeugt, in der wichtige Ereignisse wie Baubeginn, Baufehler und natürlich die Plugin-Überprüfung festgehalten werden. Suchen Sie im Datei-Explorer nach dem Protokollordner <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>:

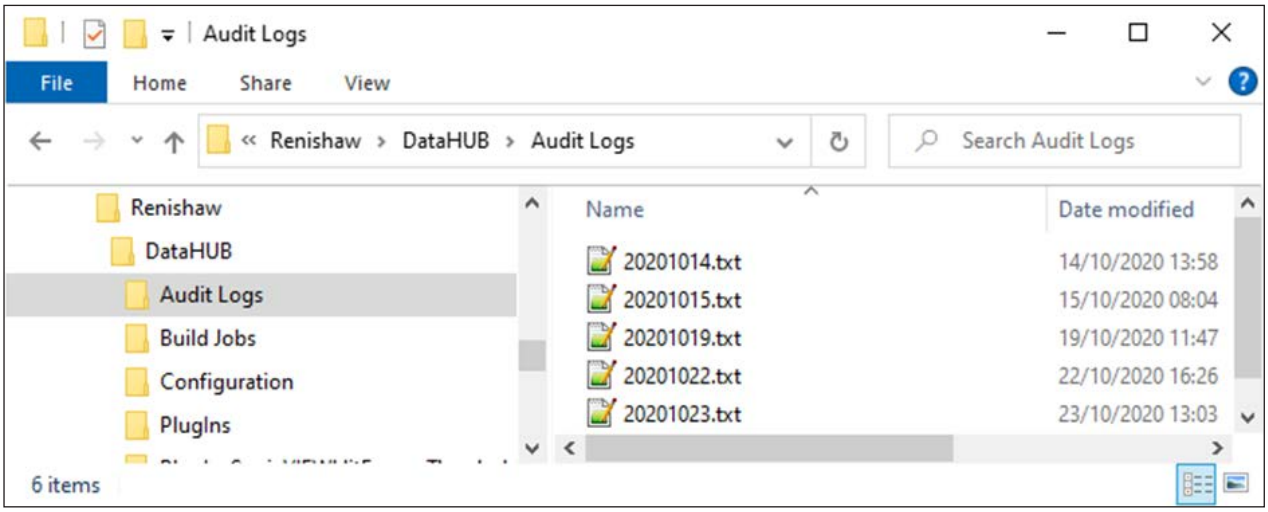


Abbildung 30 DataHUB Protokollordner

Öffnen Sie die jüngste Protokolldatei – sie sollte das aktuelle Datum aufweisen – blättern Sie bis zum Ende und suchen Sie nach Zeilen mit folgenden Angaben, um gültige Plug-ins ausfindig zu machen:

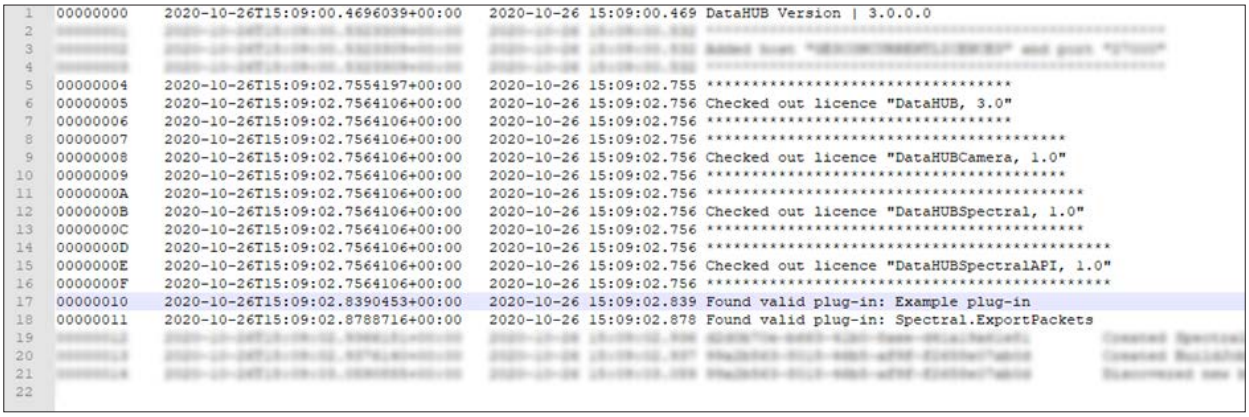


Abbildung 31 Von DataHUB gefundene gültige Plug-ins

6.4.4.13 Zeitlimit (Timeouts)

Da DataHUB nicht garantieren kann, dass auf jeden Aufruf einer Plug-in-Instanz eine zeitnahe Rückmeldung erfolgt, wird eine Timeout-Regel angewendet. Wenn ein Plug-in nicht innerhalb einer bestimmten Zeit durchgeführt wird, gilt es als fehlerhaft und wird abgebrochen. Die Standard-Timeouts sind:

- Für *Initialise*: 20 Sekunden
- Für *Process* (wie auch *ProcessPackets* in API V1.0), 200 Sekunden
- Für *Shutdown*: 20 Sekunden

Beim Debuggen werden diese Timeouts häufig überschritten und es kann vorkommen, dass DataHUB das Plug-in vorzeitig beendet. Um mehr Zeit für das Debuggen zu ermöglichen, können die Timeout-Limits in der Datei <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> mit einem geeigneten XML- oder Texteditor erhöht werden.

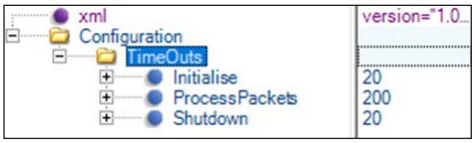


Abbildung 32 Standard-Timeouts für Plug-ins

HINWEIS: Damit Timeout-Änderungen wirksam werden, muss der DataHUB Service neu gestartet werden.

6.4.4.14 Plug-in debuggen

Nachdem das Plug-in erfolgreich eingefügt wurde, ist es nun möglich, seine Ausführung zu debuggen. Das Plug-in selbst ist nicht direkt ausführbar, aber wenn Sie den Visual Studio Debugger an den DataHUB Service anfügen und einen Bauauftrag starten, ruft der Service die Plug-in-Implementierung auf, die an den im Code vorgesehenen Haltepunkten gestoppt wird.

HINWEIS: Zum Debuggen über den DataHUB Service benötigt Visual Studio möglicherweise Administratorrechte. Klicken Sie dazu mit der rechten Maustaste auf die Visual Studio-Programmverknüpfung und wählen Sie **Run as administrator** (Als Administrator ausführen).

Fügen Sie in Visual Studio zu Beginn der Methoden *Initialise*, *Process* und *Shutdown* Haltepunkte ein:

Abbildung 33 Haltepunkte in den Methoden *Initialise* und *ProcessPackets*

Öffnen Sie das Menü **Debug** (Debuggen) und wählen Sie **Attach to process…** (An Prozess anfügen):

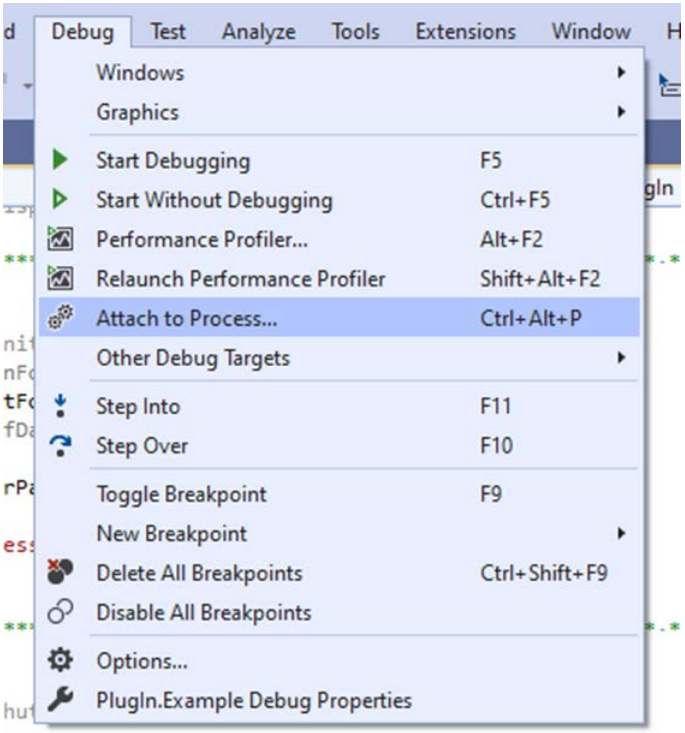


Abbildung 34 Zum Debuggen an einen Prozess anfügen

Suchen Sie in der Liste der laufenden Prozesse die Datei *DataHUB Service.exe*. Möglicherweise müssen Sie die Option **Show processes from all users** (Prozesse aller Benutzer anzeigen) aktivieren.

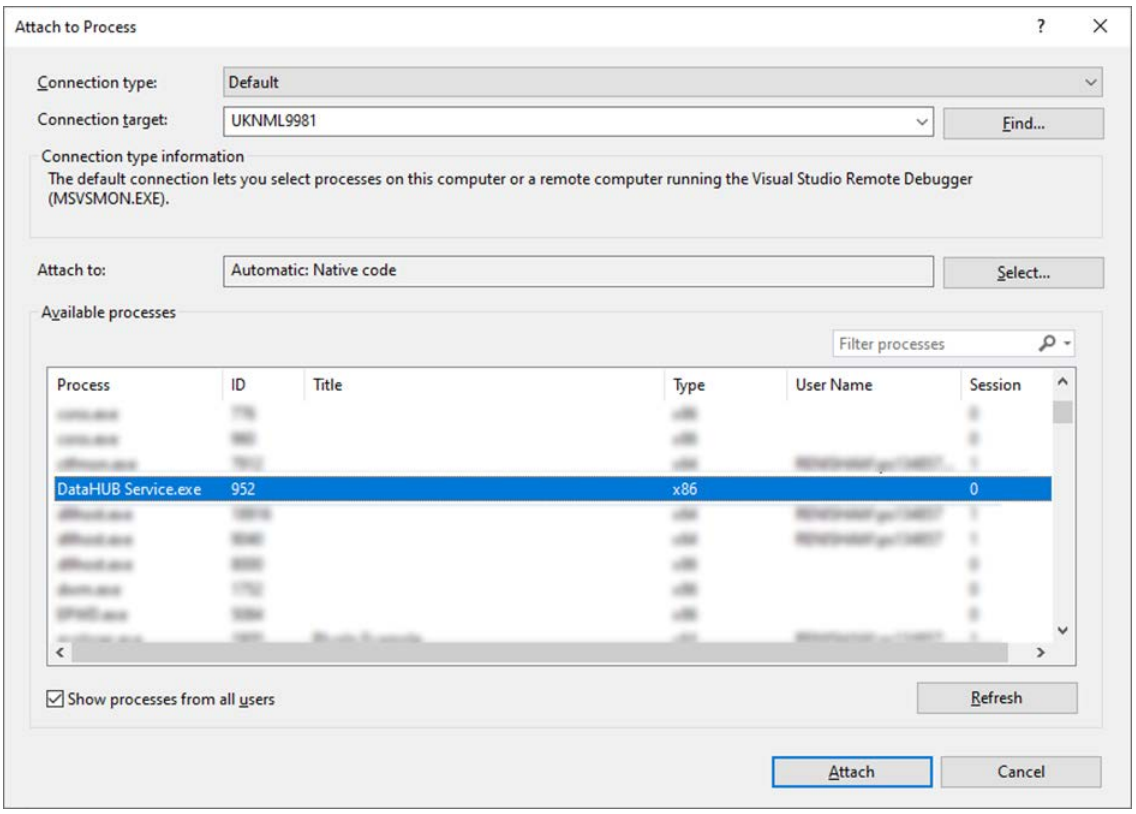


Abbildung 35 DataHUB Service Prozess

Drücken Sie die Schaltfläche **Attach** (Anfügen). Nun beginnt Visual Studio beginnt eine Debugging-Session für den DataHUB Service.

Damit DataHUB das Plug-in aufruft und somit die Haltepunkte getroffen werden können, muss ein Bau gestartet werden. Dies kann über den DataHUB Generator erfolgen. Wenn Sie bereits einige Bauvorgänge generiert haben, können Sie aber auch einfach einen Bauordner in <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs> duplizieren.

DataHUB erkennt den neuen Bau automatisch und beginnt mit der Verarbeitung der Daten. Eine der ersten Aufgaben des Dienstes besteht darin, eine Instanz jedes registrierten Plug-ins zu erstellen (durch Aufrufen des parameterlosen Konstruktors). Wenn DataHUB dann die Instanz der Initialisierungsmethode *Initialise* aufruft, ist der erste Haltepunkt erreicht. Die Methodenparameter werden durch die Werte des betreffenden Baus ersetzt, zum Beispiel:

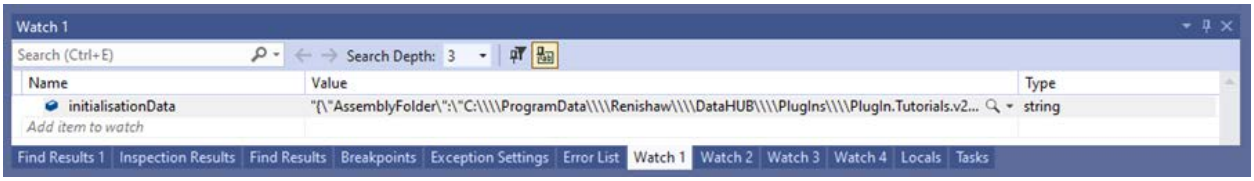


Abbildung 36 Parameterwerte der Initialise-Methode

Von diesem Punkt an werden vorhandene Daten (bei einem laufenden Bau auch neue Daten, sobald sie eintreffen) verarbeitet, und die *Process*-Methode der Instanz wird mit Parameterwerten für die LaserVIEW- und MeltVIEW-Daten aufgerufen, die einer Schicht zugeordnet sind:

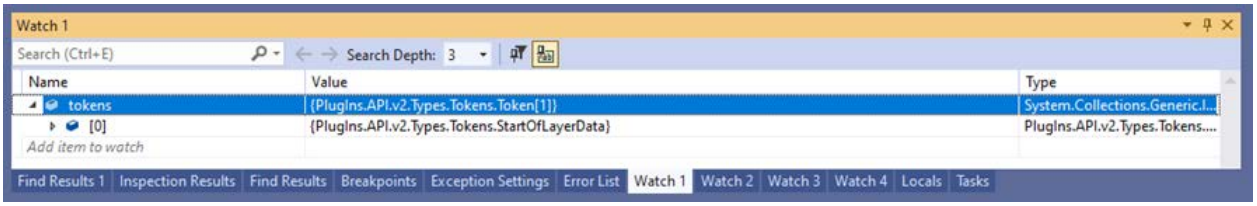


Abbildung 37 Parameterwerte der Process-Methode

6.4.4.15 Anwendung des Plug-ins in der Produktion

Um die Anwendung des Plug-ins auf einem Produktions-DCPC vorzubereiten, muss ein *Release*-Build kompiliert, lokal eingesetzt und getestet werden. Sobald die Funktionalität des Plug-ins validiert und verifiziert wurde, kann es auf Produktions-DCPCs eingesetzt werden.

Die Bereitstellung auf einem Produktions-DCPC verläuft ähnlich wie die Bereitstellung auf einem Entwicklungs-PC. Der Ordner *Release* sollte in den Unterordner *PlugIns* im DataHUB Datenordner (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) auf dem DCPC kopiert und passend zum dem Namen des Plug-ins umbenannt werden. Der DataHUB Service muss dann neu gestartet werden, um das neue Plug-in zu registrieren. Verwenden Sie hierzu die *Services* Anwendung. Die erfolgreiche oder fehlgeschlagene Registrierung des neuen Plug-ins wird im Audit-Log protokolliert.

HINWEIS: DataHUB ist so konzipiert, dass alle laufenden Datenverarbeitungsaufgaben nach einem Neustart wieder aufgenommen werden. Es ist also nicht notwendig, vor dem Neustart zu warten, bis alle laufenden Aufgaben abgeschlossen sind. Allerdings wird das neu registrierte Plug-in nur für neue Datenverarbeitungsaufgaben angewendet.

6.4.4.16 Diagnose von Plug-in-Fehlern in der Produktion

In der Produktionsumgebung zeigt DataHUB Monitor (neben anderen Verarbeitungsaufgaben) den Verarbeitungsstatus für das Plug-in an:

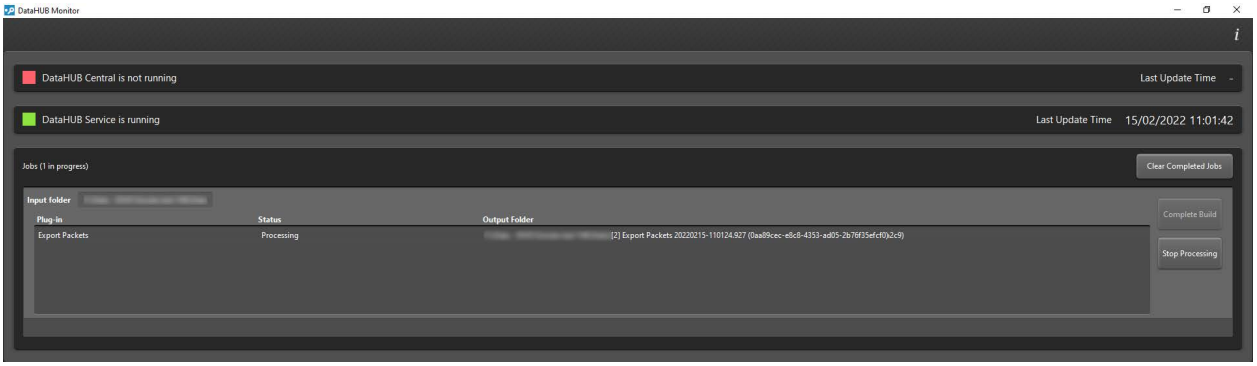


Abbildung 38 Laufender Bau mit Plug-ins

Lautet der Status eines Plug-ins *Error* (Fehler), so wird dieser Fehler im Audit-Protokoll erfasst. Dies erfolgt entweder in Form von Einträgen, die von DataHUB selbst hinzugefügt wurden (z. B. fehlgeschlagenes Laden der Plug-in-Assembly aufgrund fehlender Abhängigkeiten), oder durch das Plug-in über die Ausgabewerte der Plug-in-Methoden *Initialise*, *Process* oder *Shutdown*.

6.4.5 Beispiel 1 – Verarbeitung eines Token-Streams

Dieses erste Code-Beispiel zeigt, wie ein Token-Stream verarbeitet wird, um die Mindest- und Höchstwerte für jeden Kanal der Prozessüberwachungsdaten, für jeden Laser und für jede Schicht zu extrahieren. Hier werden mehrere Schlüsselkonzepte besprochen:

- Entschlüsselung der Initialisierungsdaten
- Verarbeitung von Tokens
- Ausgabeordner der Instanz

6.4.5.1 Erstellung

Die minimale Plugin-Implementierung sieht wie folgt aus:

```
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();
   }
}
```

Die API-Helferklassen `InitialiseReturns`, `ProcessReturns` und `ShutdownReturns` bieten bequeme Methoden zur Erzeugung von API-Rückgabewerten. Hier geben die *Success*-Methoden den von den DataHUB erkannten Standardstring „Success“ (Erfolg) zurück.

6.4.5.2 Initialisierung

Wenn DataHUB dieses Plug-in initialisiert, besteht die erste Aufgabe darin, den Ort des eindeutigen Ausgabeordners für die Plug-in-Instanz zur späteren Verwendung beim Schreiben in eine Datei zu speichern. Die Methode entschlüsselt (anhand der Newtonsoft.Json- Bibliothek) den String „`initialisationData`“ und extrahiert den Pfad des Ausgabeordners:

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

HINWEIS: Die Initialisierungsdaten werden als JSON-formatierter String kodiert.

6.4.5.3 Verarbeitung des Token-Streams

Bei erfolgreicher Initialisierung erhält die Plug-in-Instanz den Token-Stream des Baus. Da manche Prozessüberwachungsdatensätze sehr groß sein können, wird der Token-Stream in Stapeln geliefert und bei jedem Aufrufen von `Process` der nächste Token-Stapel empfangen.

Nun kann die Anweisung zur Bearbeitung der `Sample`-Tokens der `Process`-Methode hinzugefügt werden:

```
…
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);
…
```

Es werden mehrere Felder initialisiert, um das laufende Minimum und Maximum für jede Eigenschaft der Prozessüberwachungsdaten eines `Sample`-Tokens zu speichern.

Beachten Sie, dass sich die Rückgabe aus der `Process`-Methode zu `SuccessWithoutInformation()` geändert hat. Dies weist DataHUB darauf hin, dass das Plug-in die Verarbeitung dieses Token-Stapels beendet hat, aber keine Meldung in die DataHUB Protokolle einfügen möchte. Auf diese Weise werden die DataHUB Protokolle nicht mit trivialen Erfolgsmeldungen überfrachtet.

HINWEIS: Die Token-Streams werden in Stapeln verarbeitet; die `Process`-Methode wird mehrmals aufgerufen.

HINWEIS: Nicht jeder Aufruf von `Process` muss eine protokollierte Meldung an den DataHUB Service zurückgeben.

Der Wert jeder Eigenschaft der Prozessüberwachungsdaten des `Sample`-Tokens wird verwendet, um das laufende Minimum und Maximum zu aktualisieren. Beachten Sie, dass manche Eigenschaften den Typ „double“ und andere den Typ „integer“ haben.

HINWEIS: Die Eigenschaften der Prozessüberwachungsdaten für ein `Sample`-Token lauten wie folgt:

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

6.4.5.4 In Datei schreiben

Die letzte Aufgabe des Plug-ins besteht darin, bei der Verarbeitung eines `EndOfLayerLaserData` Tokens die Ergebnisse in eine Datei zu schreiben:

```
…
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}";
}
…
```

Der Ausgabepfad für die Ergebnisdatei wird aus einer Kombination des bei der Initialisierung zwischengespeicherten Ausgabeordners und der Schicht- und Lasernummern für die soeben verarbeiteten Tokens erstellt. Die Mindest- und Höchstwerte für jede `Sample`-Eigenschaft werden in die Datei geschrieben und dann zur Verarbeitung der Tokens für die nächste Schicht zurückgesetzt.

Das Plug-in liefert eine Erfolgsmeldung („Success“), die von DataHUB geprüft wird.

HINWEIS: DataHUB weist beim Ausführen jeder Plug-in-Instanz einen eindeutigen Ausgabeordnerpfad zu, der zum Speichern der Plug-in-Ausgabe verwendet werden sollte.

6.4.5.5 Abschließende Implementierung

Die vollständige Implementierung des Plug-ins wird folgendermaßen durchgeführt:

```
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 Filter

Ein *Filter* in der API ist eine Codeeinheit, die ein Eingabetoken verarbeitet und null oder mehr Tokens ausgibt. Ein Filter kann das Eingabetoken nach der entsprechenden Verarbeitung ausgeben, neue Tokens ausgeben oder das Eingabetoken sogar schlucken. Mit diesen Methoden wandelt ein Filter einen Eingabetoken-Stream in einen neuen Ausgabetoken-Stream um.

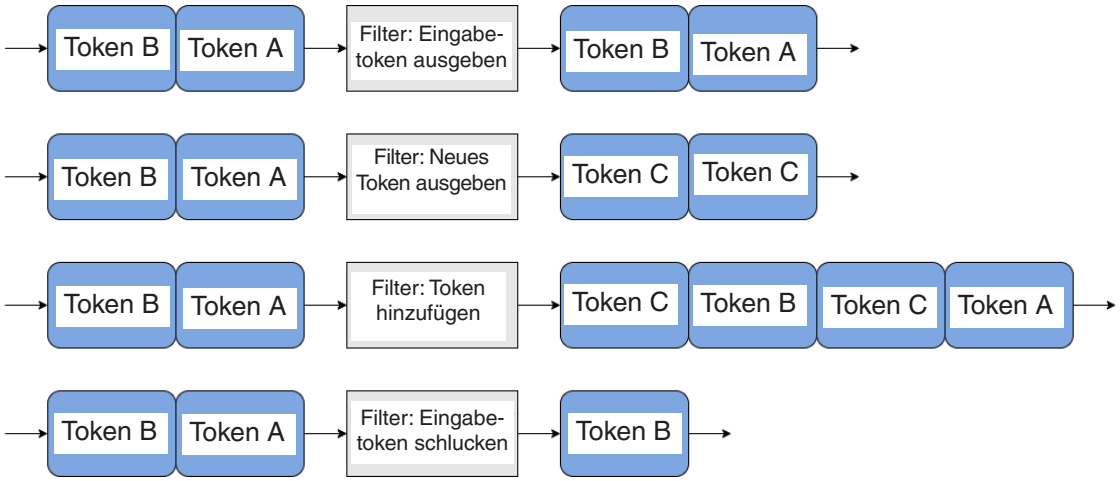


Abbildung 39 Filterausgabemethoden

6.4.6.1 Pipelines

Die API bietet die Möglichkeit, zwei Filter zu einer *Pipeline* zu kombinieren. Der Umstand, dass eine Pipeline die gleiche Signatur wie ein einzelner Filter hat, bedeutet, dass Pipelines und Filter zu längeren, komplexeren Pipelines kombiniert werden können. Wenn eine Pipeline einen Token-Stream verarbeitet, bildet die Ausgabe des ersten Filters die Eingabe für den zweiten, und so weiter, bis die Ausgabe des letzten Filters die abschließende Ausgabe der Pipeline bildet.

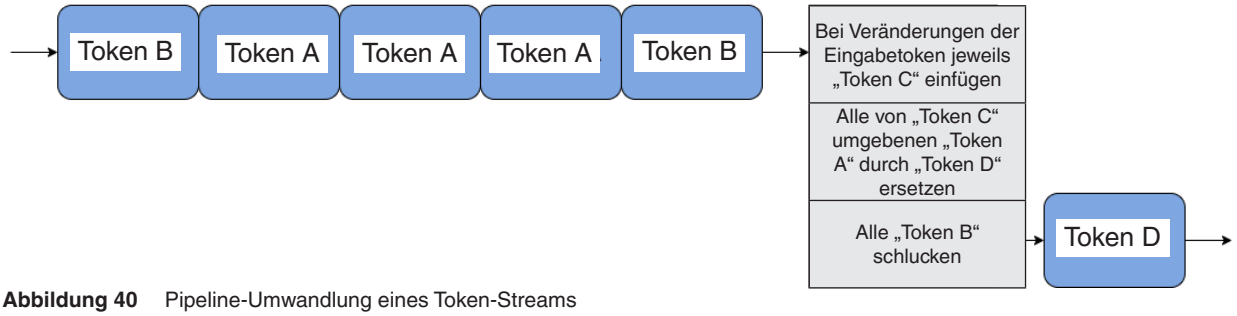


Abbildung 40 Pipeline-Umwandlung eines Token-Streams

6.4.6.2 Zusammenfassen von Samples zu Paketen unter Verwendung einer Filterpipeline

Dieser Abschnitt veranschaulicht die Umwandlung eines Eingabetoken-Streams der erfassten Samples über mehrere Filterstufen in einen Ausgabetoken-Stream mit zusammengefassten Daten.

Die Pipeline wird aus den folgenden Filtern gebildet:

- Splitten, wenn sich die Lasermodulation ändert
- Senden, wenn der Laser feuert
- Splitten, wenn sich die angeforderte X-Position ändert
- Splitten, wenn sich angeforderte Y-Position ändert
- Zusammenfassen in Sample-Pakete
- Sample-Pakete zusammenfassen

6.4.6.2.1 Eingabetoken-Stream

Der Eingabetoken-Stream besteht aus sieben Beispieltokens:

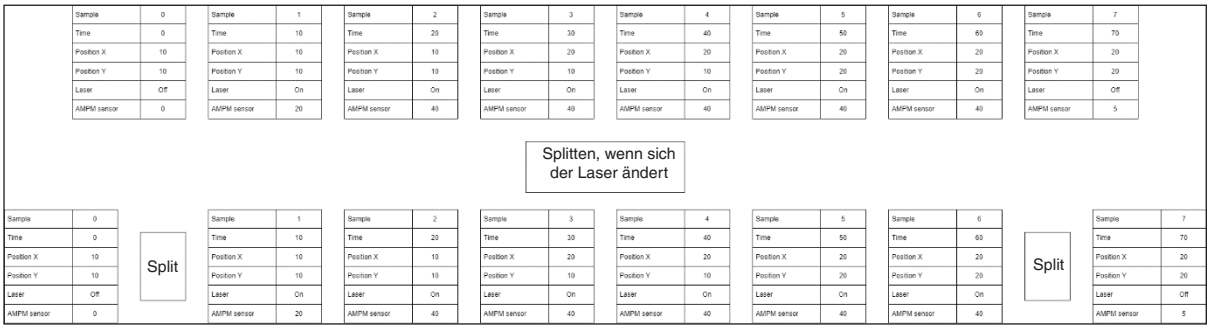
Sample	0	Sample	1	Sample	2	Sample	3	Sample	4	Sample	5	Sample	6	Sample	7
Time	0	Time	10	Time	20	Time	30	Time	40	Time	50	Time	60	Time	70
Position X	10	Position X	10	Position X	10	Position X	20	Position X	20	Position X	20	Position X	20	Position X	20
Position Y	10	Position Y	10	Position Y	10	Position Y	10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

Die Funktionsweise des Lasers kann anhand der verschiedenen Werte in jedem Sample nachvollzogen werden:

- Der Laser beginnt an der Position (10, 10) zum Zeitpunkt 0 und feuert nicht
- Der Laser feuert am Zeitpunkt 20
- Der Laser bewegt sich zum Zeitpunkt 30 zur Position (20, 10)
- Der Laser bewegt sich zum Zeitpunkt 50 zur Position (20, 20)
- Der Laser stoppt am Zeitpunkt 70 zu feuern

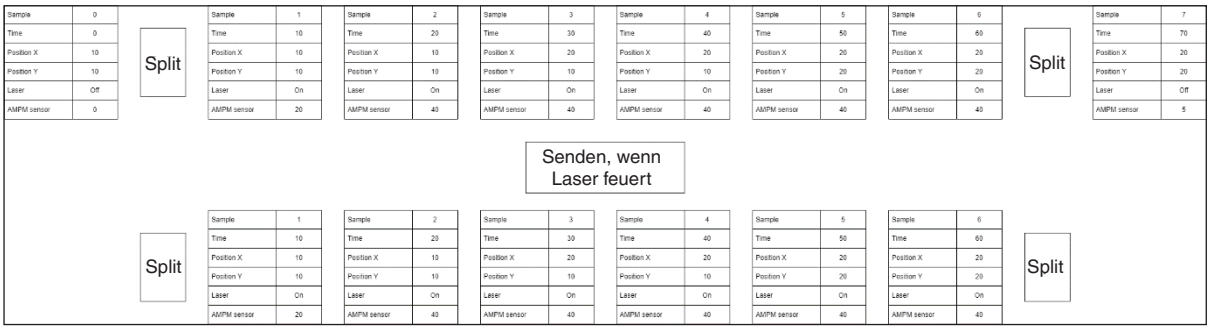
6.4.6.2.2 Filter 1: Splitten, wenn sich die Lasermodulation ändert

Der erste Filter fügt `Split`-Tokens zwischen geänderten Lasermodulationskanälen ein:



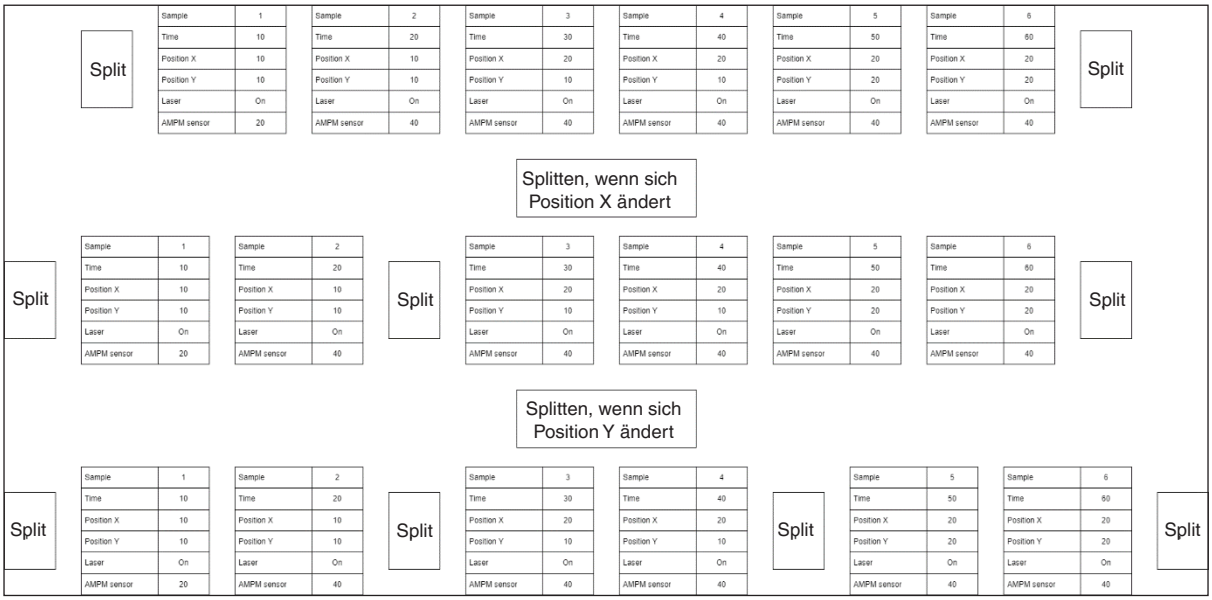
6.4.6.2.3 Filter 2: Senden, wenn Laser feuert

Mit dem nächsten Filter werden alle Samples entfernt, bei denen der Laser ausgeschaltet war:



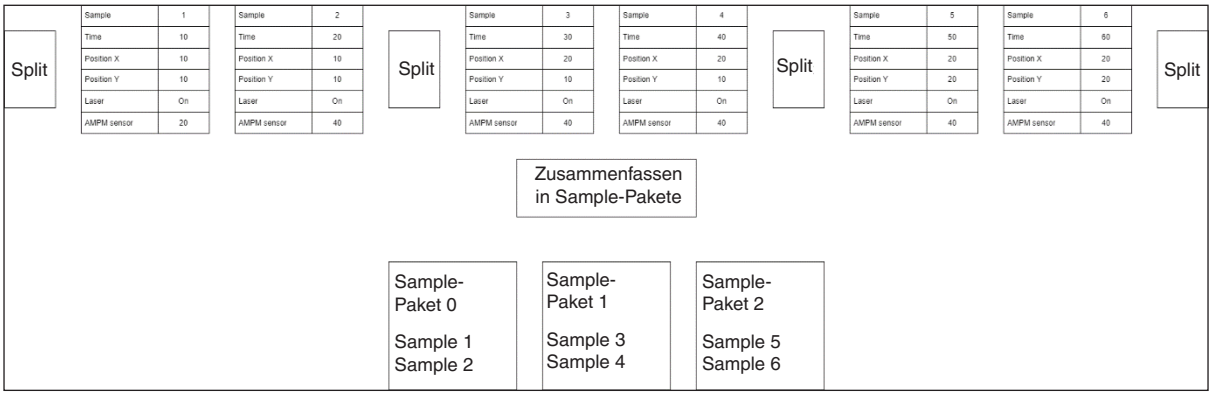
6.4.6.2.4 Filter 3 und 4: Splitten, wenn sich die Position ändert

Die nächsten beiden Filter fügen `Split`-Tokens zwischen Samples ein, bei denen sich die Laserposition geändert hat:



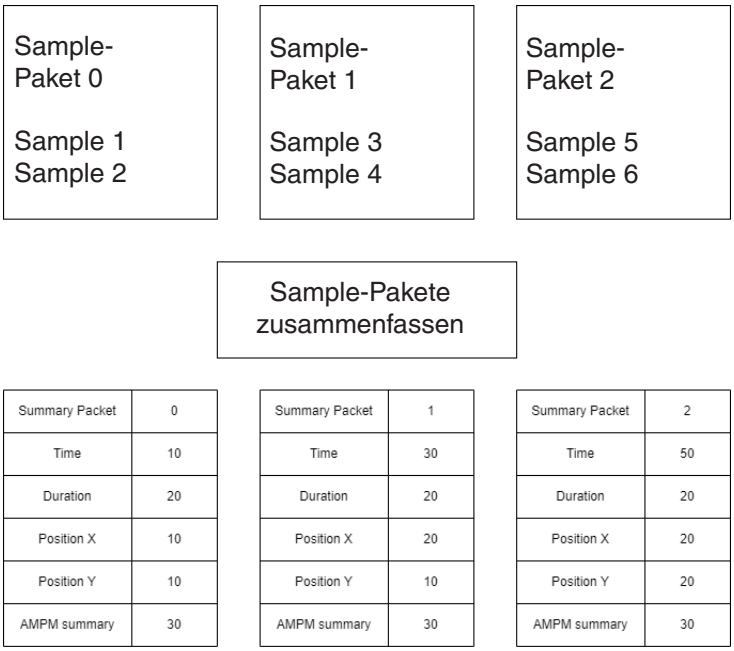
6.4.6.2.5 Filter 5: Zusammenfassen in Sample-Pakete

Die Samples werden nun von `Split`-Tokens umgeben. Zwischen den Tokens befinden sich jeweils die Samples, bei denen der Laser gezündet hat und stationär war – d. h., die Samples für ein einziges Schmelzbad. Diese Gruppen von Samples werden dann zusammengefasst.



6.4.6.2.6 Filter 6: Sample-Pakete zusammenfassen

Abschließend werden die gruppierten Samples zusammengefasst:



Jedes dieser abschließenden, zusammenfassenden Paket-Tokens enthält Werte für die Startzeit (Zeit des ersten beinhalteten Samples), die Dauer (Anzahl der beinhalteten Samples × Dauer eines einzelnen Samples), die Position und zusammenfassende Prozessüberwachungswerte.

6.4.7 Beispiel 2 – Filter

In diesem Beispiel wird das Minimum/Maximum-Plug-in mithilfe der in der API bereitgestellten Filter erweitert. Folgende Schlüsselkonzepte werden hier besprochen:

- Filter
- Verarbeitung von Tokens durch einen Filter

6.4.7.1 Filter

Filter verarbeiten und verändern einen Token-Stream, indem sie Tokens hinzufügen, entfernen, weiterleiten oder verändern. Ein Filter verarbeitet ein einzelnes Token und gibt null oder mehr Tokens aus. Diese Zuordnung nach dem „One-to-Many“-Prinzip bietet einen leistungsfähigen Mechanismus für das Ändern eines Token-Streams während der Verarbeitung.

Bei vielen Analyseaufgaben im Rahmen der Prozessüberwachung sind nur Samples von Interesse, die aufgezeichnet wurden, als der Laser zündete (da sie Schmelzbaddaten enthalten). Das Herausfiltern von Samples, die aufgezeichnet wurden, während der Laser nicht zündete, reduziert überdies die Anzahl der Tokens, die bei der Analyse berücksichtigt werden müssen. Der eingebaute Filter `EmitSampleIf.LaserModulate.IsOn` erfüllt diesen Zweck. Das Plug-in kann jedes `Sample`-Token durch diesen Filter leiten, um alle Samples auszufiltern, bei denen der Laser ausgeschaltet war, und liefert so die für die Schmelzbadbildung relevanten Mindest- und Höchstwerte.

In C# ist ein Delegat ein Typ, der eine Methode mit einer bestimmten Signatur darstellt. Der Wert eines Delegaten wird mit einer *Methodeninstanz* zugewiesen, und diese Methode kann über den Delegaten aufgerufen werden.

Ein neuer Namespace `PlugIn.Tutorials.v2.Example2` und eine Plug-in-Klasse werden mit dem Delegaten `FilterOutLaserOffSamples` hinzugefügt, dem beim Erzeugen der Plug-in-Instanz der Filter `EmitSampleIf.LaserModulate.IsOn()` zugewiesen wird:

```
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();
  }
}
```

Die Process-Methode wird als Nächstes implementiert. Dabei wird ein LINQ-Ausdruck verwendet, um jeden Token-Stapel zunächst durch den Filter (über den Delegaten) und dann durch eine Methode zu leiten, die den resultierenden geänderten Token-Stream verarbeitet.

```
…
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}";
        }
}
```

HINWEIS: Aus Platzgründen werden die verschiedenen Felder (`_demandX`, etc.) und Helfermethoden (`FormatMinAndMax` usw.) nicht gesondert erwähnt. Sie werden wie im ersten Beispiel implementiert.

Der LINQ-Ausdruck besteht aus zwei Stufen: Zuerst wird jeder Token-Stapel mittels einer API Helfermethode namens `FilteredBy` zum Filter `FilterOutLaserOffTokens` geleitet. Dann werden die Tokens im resultierenden Token-Stream (wobei `Sample`-Tokens nur Samples erfassen, bei denen der Laser feuerte) zu `ProcessToken` geleitet. Hier wird die spezielle Verarbeitung für jeden relevanten Token-Typ durchführt, d. h. `Sample` und `EndOfLayerLaserData`. Die von `ProcessTokens` zurückgegebenen resultierenden Strings werden gesammelt und über die `Success`-Helfermethode an DataHUB zurückgegeben.

HINWEIS: Die LINQ-Syntax von C# ist auf die Verarbeitung von Datenströmen ausgelegt.

HINWEIS: Token-Streams können durch einen Filter modifiziert werden.

HINWEIS: Ein Filter gibt für jedes Eingabetoken null oder mehr Tokens zurück.

Das Plug-in füllt nun die Ergebnisdateien mit den Mindest- und Höchstwerten des Kanals für die Samples, die erfasst wurden, während der Laser in Betrieb war.

6.4.7.2 Abschließende Implementierung

Die vollständige Implementierung des Plug-ins wird folgendermaßen durchgeführt:

```
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 Beispiel 3 – ExportPackets

Einzelne Filter sind zwar nützlich, aber am leistungsfähigsten sind sie, wenn man sie in einer Pipeline kombiniert (siehe „Pipelines“ auf Seite 6-56). Die Pipeline nimmt dann mithilfe einer Reihe von einfachen, wiederverwendbaren Filtern komplexe Operationen am Token-Stream vor. Genau dies tut das `ExportPackets` Plug-in, das mit dem DataHUB Service bereitgestellt wird. Dieses Beispiel zeigt eine Implementierung des Plug-ins, einschließlich:

- Kombinieren von Filtern in einer Pipeline
- Verarbeitung eines Token-Streams durch eine Pipeline

6.4.8.1 Kombinieren von Filtern

Zum Kombinieren von Filtern in eine Pipeline stellt die Plug-in API die beiden Helfer `First` und `Then` bereit. Daraus wird eine Pipeline folgendermaßen erzeugt:

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

Eine beliebige Anzahl von Filtern kann miteinander verkettet werden, und die resultierende Pipeline verhält sich wie ein Filter, der ein einziges Eingabetoken zu null oder mehr Ausgabetokens zuordnet.

Das mit DataHUB bereitgestellte Plug-in *ExportPackets* verarbeitet einen Token-Stream in Pakete. Dabei ist ein Paket eine Sammlung von Prozessüberwachungsdaten, die für aufeinanderfolgende Samples gesammelt wurden, während ein Laser an einer bestimmten, angeforderten X- und Y-Position feuerte. Um Samples zu Paketen zusammenzufassen, wird ein Token-Stream wie folgt verarbeitet:

1. Splitten Sie den Stream jedes Mal, wenn sich der Ein-/Aus-Status des Lasers ändert.
2. Entfernen Sie alle Samples, bei denen der Laser nicht feuerte.
3. Splitten Sie den Token-Stream jedes Mal, wenn sich die X-Position ändert.
4. Splitten Sie den Datenstrom jedes Mal, wenn sich die Y-Position ändert.
5. Fassen Sie jede durch Split-Tokens eingegrenzte Gruppe an Samples zu einem Sample-Paket zusammen.
6. Erzeugen Sie für jedes Sample-Paket einen Mittelwert sowie einen Medianwert.

HINWEIS: Samples im Stream, bei denen der Laser nicht aktiv war, werden nicht zuerst herausgefiltert, wie es auf den ersten Blick sinnvoll erscheinen mag, da dies zu fehlerhaften Paketen führen würde, wenn der Laser beim wiederholten Ein- und Ausschalten auf derselben angeforderten X- bzw. Y-Position verbleibt.

Der Startcode für das Plug-in lautet:

```
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;
    }
}
```

Das Plug-in verwendet einen `StringBuilder`, um die Zusammenfassungen aller Sample-Pakete zu erstellen. Zuerst fügt es die Überschriften für jede Spalte in der csv-Datei hinzu.

Die oben beschriebene Pipeline wird erstellt und dem Delegaten `CollateIntoPackets` zugewiesen:

```
…
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;
}
…
```

Die Pipeline besteht aus mehreren integrierten Filtern, die so verkettet sind, dass die endgültige Ausgabe der Pipeline eine Reihe von `SummaryPacket`-Tokens mit Mittel- und Medianwerten enthält. Einen Überblick über die Funktion der Pipeline finden Sie unter „Zusammenfassen von Samples zu Paketen unter Verwendung einer Filterpipeline“ auf Seite 6-57.

HINWEIS: Einfache Filter können miteinander in einer Pipeline verkettet werden, um einen Eingabetoken-Stream einer komplexen Transformation zu unterziehen.

Der letzte Filter, `SummarisePacketOfSamples.Summarise`, wird mit `SummaryMethod`- Anweisungen initialisiert. Sie geben die Art der Zusammenfassung, d. h. „Mean“ (Mittelwert) bzw. „Median“, sowie wie die Methode zur Berechnung des Werts vor. Auf diese Weise lässt sich flexibel konfigurieren, wie Sample-Pakete zusammengefasst werden sollen. Sie können verschiedene Zusammenfassungsmethoden verwenden, um beispielsweise Mittelwerte, Mediane, Mindest- und Höchstwerte zu ermitteln:

```
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)))
```

Der letzte Schritt ist die Implementierung der *Process*-Methode. Dabei wird dem `StringBuilder` (mit der Helfermethode `AddSummaryLine`) eine Zeile für jedes `SummaryPacket`-Token hinzugefügt und der Inhalt von `StringBuilder` in die Ausgabedatei geschrieben:

```
…
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);
}
…
```

Jedes `SummaryPacket`-Token wird – mit korrekt formatierten Werten – mit dem `StringBuilder` als Textzeile hinzugefügt. Wenn ein `EndOfLayerLaserData`-Token angetroffen wird, schreibt das Plug-in die zusammengefassten Paketdaten in eine Datei im Ausgabeordner.

HINWEIS: Der Token-Stream für den Bau kann über eine vordefinierte Pipeline verarbeitet werden.

6.4.8.2 Abschließende Implementierung

Die vollständige Implementierung des Plug-ins wird folgendermaßen durchgeführt:

```
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 Beispiel 4 – Rückgabe von Zeitseriendaten an Renishaw Central

Das folgende Beispiel zeigt, wie ein Plug-in eine Renishaw Central Zeitreihe erstellt und mit Datenpunkten füllt. Die Renishaw Central Web-UI zeigt Zeitreihendaten als Diagramm an.

Das Beispiel-Plug-in definiert eine Zeitreihe und fügt für jede Schicht einen Datenpunkt für den Höchstwert (Maximum) des LaserVIEW-Kanals hinzu. Die Definition der Zeitreihe und ihre Datenpunkte werden über die Rückgabewerte der Methoden *Initialise* und *Process* zurückgegeben.

Folgende Schlüsselkonzepte werden hier besprochen:

- Erstellen einer Renishaw Central Zeitreihe
- Hinzufügen von Daten zu einer Renishaw Central Zeitreihe

Wir beginnen mit dem leeren Standard-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 Initialisierung

Bei der Initialisierung dieses Plug-ins durch DataHUB besteht die Aufgabe darin, eine Definition für die Zeitreihe „LaserVIEW Max“ zurückzugeben. Eine Zeitreihendefinition erfordert einen Namen, einen Einheitentyp und eine Einheit. Daneben können zusätzliche Eigenschaften definiert werden, z. B. die Eigenschaft „Grouping“. Sie ordnet die Zeitreihe in eine Gruppierung ein, die das Renishaw Central Web-Frontend zur Anzeige und Verwaltung verwandter Zeitreihen verwendet.

Die Zeitreihendefinition wird mit der `InitialiseReturns`-Methode `SuccessAndCreateTimeSeries()` an DataHUB zurückgegeben. DataHUB konvertiert den zurückgegebenen Wert, um Renishaw Central die erforderlichen Daten zur Erstellung der Zeitreihe zu liefern, die dann mit Datenpunkten gefüllt werden kann:

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

HINWEIS: Die Initialisierungsmethode des Plug-ins kann eine oder mehrere Zeitreihendefinitionen zurückgeben.

HINWEIS: DataHUB verwaltet die Übermittlung von Daten an Renishaw Central, sodass sich die Plug-ins auf die Zeitreihendefinitionen konzentrieren können.

6.4.9.2 Verarbeitung

Bei der Verarbeitung jedes `Sample`-Tokens im Token-Stream wird der aktuelle Höchstwert (max.value) mit dem LaserVIEW-Kanalwert des Samples verglichen und ggf. aktualisiert.

```
…
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);
…
```

Wenn das Plug-in am Ende der Laserdaten für die aktuelle Schicht das Token `EndOfLayerLaserData` empfängt, kann eine Erfolgsmeldung an DataHUB zurückgegeben werden, um zu signalisieren, dass die Daten für die angegebene Schicht und den angegebenen Laser verarbeitet wurden:

```
…
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();
}
…
```

Am Ende der Schichtdaten erhält das Plug-in ein `EndOfLayerData`-Token. Das `TimeSeries`-Objekt wird verwendet, um ein `UpdateTimeSeries`-Objekt mit einem Zeitserienwert zu erzeugen, der den abschließenden Höchstwert der Schicht enthält. Die `ProcessReturns` Helfer stellen die Methode `SuccessAndSendData()` bereit, die ein Array von `UpdateTimeSeries`-Objekten aufnimmt.

Der zwischengespeicherte Höchstwert für die Schicht wird für den nächsten Token-Stapel zurückgesetzt, und die aktualisierte Zeitreihe wird zurückgegeben:

```
…
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 Abschließende Implementierung

Die vollständige Implementierung des Plug-ins wird folgendermaßen durchgeführt:

```
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 Beispiel 5 – Auslösen von Warnmeldungen in Renishaw Central

Dieses Beispiel veranschaulicht, wie Meldungen in Renishaw Central ausgelöst werden können, wenn bestimmte Werte die vorgegebenen Grenzen überschritten haben.

Folgende Schlüsselkonzepte werden hier besprochen:

- Entschlüsselung der Initialisierungsdaten
- Auslösen von Meldungen

6.4.10.1 Initialisierungsdaten für Plug-in-Auftrag konfigurieren

Die Grenzwerte, deren Überschreitung in diesem Beispiel Meldungen auslöst, werden über die Initialisierungsdaten des Plug-ins gesteuert. In diesem Beispiel werden die Initialisierungsdaten entsprechend der folgenden Struktur im Schritt **Specify Plug-in Initialisation Data** (Plug-in-Initialisierungsdaten angeben) von DataHUB Generator eingegeben.

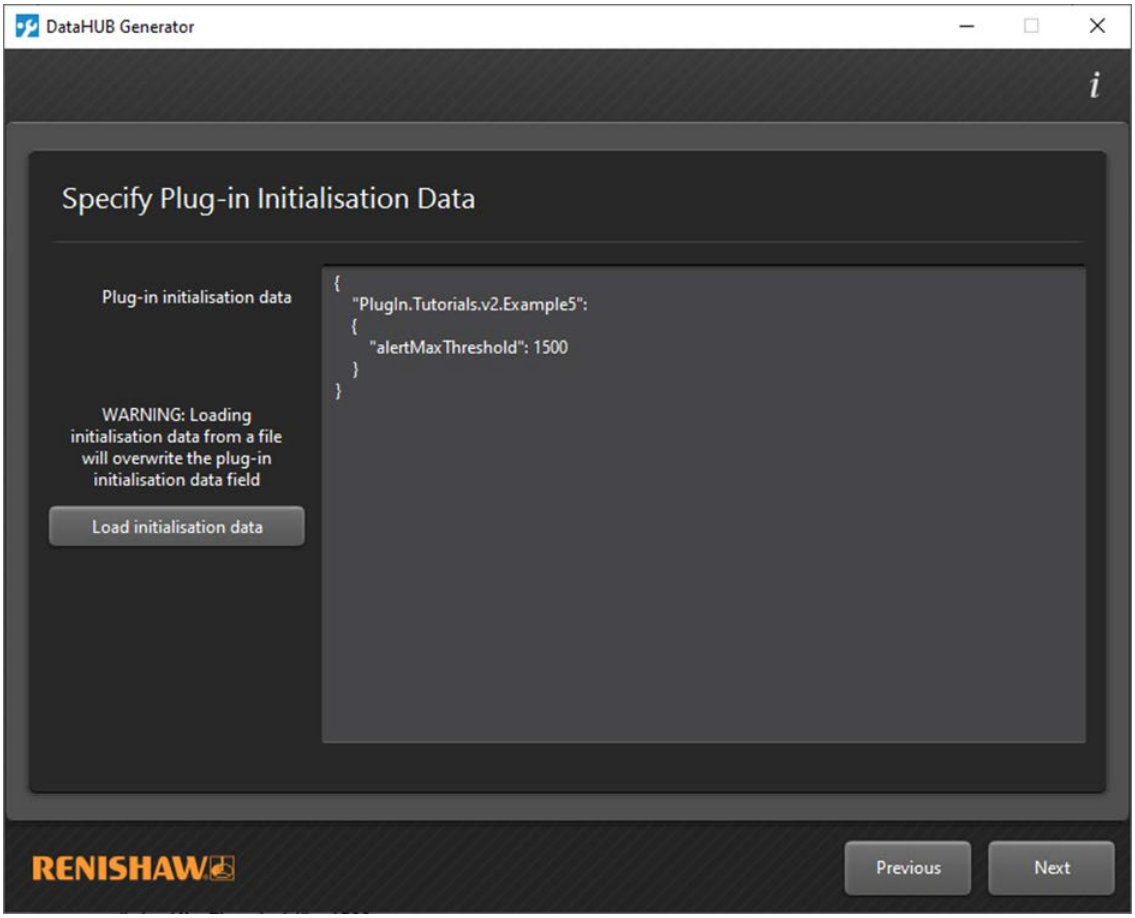


Abbildung 41 Initialisierungsdaten für Plug-in über den DataHUB Generator eingeben

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

Wir beginnen mit dem leeren Standard-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 Initialisierung

Bei der Initialisierung sind die Plug-in-spezifischen Initialisierungsdaten über das JSON-Objekt „initialisationData“ mit dem Namen, der bei der Konfiguration des Auftrags angegeben wurde, zugänglich (in diesem Beispiel „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;
…
```

HINWEIS: Der Plug-in-Initialisierungsstring, der beim Einrichten der Verarbeitungsaufgabe im DataHUB Generator angegeben wurde, ist für das Plug-in während der Initialisierung verfügbar.

6.4.10.3 Verarbeitung

Der Zweck dieses Plug-ins besteht darin, in Renishaw Central eine Meldung auszulösen, wenn der LaserVIEW-Höchstwert der Schicht den festgelegten Grenzwert überschreitet. Zunächst werden der aktuelle Höchstwert für die Schicht und die zugehörige Sample-Zeit zwischengespeichert und initialisiert. Bei der Verarbeitung der Sample-Tokens wird der LaserVIEW-Kanal mit dem aktuellen Höchstwert verglichen und ggf. überschrieben. Die Sample-Zeit wird ebenfalls zusammen mit dem Höchstwert gespeichert:

```
…
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);
…
```

Am Ende der Laserdaten für die jeweilige Schicht erhält das Plug-in ein `EndOfLayerLaserData`-Token. Eine Erfolgsmeldung kann an DataHUB zurückgegeben werden, um zu signalisieren, dass die Daten für die angegebene Schicht bzw. den angegebenen Laser erfolgreich verarbeitet wurden:

```
…
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();
}
…
```

Am Ende aller Samples für die Schicht erhält das Plug-in ein `EndOfLayerData`-Token, das anzeigt, dass keine weiteren Daten für diese Schicht vorliegen. Nach Erhalt dieses Tokens kann die Verarbeitung abgeschlossen und die Ergebnisse können an die Renishaw-Zentrale gesendet werden.

Wenn der aktuelle Höchstwert den definierten Grenzwert überschritten hat, wird eine Meldung erzeugt und zur Meldeliste hinzugefügt. Die Meldung wird mit einer Beschreibung, einem Schweregrad, einem Namen und einem Zeitstempel versehen.

Die `ProcessReturns`-Helfer stellen die Methode `SuccessAndSendData` bereit, die eine Meldung oder eine Reihe von Meldungen aufnimmt und diese Daten zur Rücksendung an DataHUB verschlüsselt:

```
…
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);
        …
      }
…
```

HINWEIS: An DataHUB zurückgegebene Meldungsbeschreibungen werden an Renishaw Central weitergeleitet, wo weitere Schritte wie beispielsweise der Versand einer SMS ausgelöst werden können.

6.4.10.4 Abschließende Implementierung

Die vollständige Implementierung des Plug-ins wird folgendermaßen durchgeführt:

```
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 Das Plug-in für API V2.0

Damit eine Assembly vom DataHUB Service als V2.0-Plug-in verwendet werden kann, muss die nachfolgende öffentliche API implementiert werden.

6.4.11.1 Assembly benennen

Der Dateiname der .NET Assembly muss mit „PlugIn.“ („PlugIn“ gefolgt von einem Punkt) beginnen und die Dateinamenerweiterung „dll“ haben.

6.4.11.2 Plug-in-Klasse benennen

Die Plug-in-Assembly muss eine öffentliche (public) Klasse namens PlugIn definieren.

6.4.11.3 Methoden für Plug-in-Klassen

Die Plug-in-Klasse muss die folgenden öffentlichen (public) Methoden implementieren:

```
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 Der parameterlose Konstruktor

Der Typ muss einen parameterlosen Konstruktor haben. Wenn kein Konstruktor bzw. keine Konstruktoren mit Parametern definiert sind, stellt C# diese automatisch bereit, d. h., es ist nicht notwendig, sie explizit zu definieren.

HINWEIS: Ein Plug-in sollte innerhalb seines Konstruktors keine Ressourcen akquirieren.

Bei einer Plug-in-Instanz mit Konstruktor ist nicht gewährleistet, dass ihre *Shutdown*-Methode aufgerufen wird. Solche Ressourcen werden folglich möglicherweise nicht rechtzeitig bereinigt. Ressourcen sollten über die Methode *Initialise* akquiriert werden.

6.4.11.5 Die statische `APIVersion`-Methode

Anhand der statischen Methode „`APIVersion`“ erkennt DataHUB, welche Version der API zur Validierung dieses Plug-in verwendet werden soll und welches Format die zu verarbeitenden Prozessüberwachungsdaten haben.

Parameter:

Keine.

Rückgabe: `string`

Ein Plug-in, das API V2.0 implementiert, muss das Literal „2“ zurückgeben.

6.4.11.6 Die statische `DisplayName`-Methode

Die statische `DisplayName`-Methode gibt den Namen zurück, der als Plug-in-Name in DataHUB Monitor, beim Erstellen des Ausgabeordners für das Plug-in, beim Prüfen von aufgetretenen Ereignissen usw. angezeigt wird. Dieser Name muss zwar nicht zwingend eindeutig sein, dies erleichtert jedoch das Identifizieren des Plug-ins, wenn z. B. mehrere Versionen auf einem DCPC installiert sind.

Parameter:

Keine.

Rückgabe: `string`

Das Literal, das zur Benennung des Plug-ins in verschiedenen DataHUB Benutzerschnittstellen und Protokollen verwendet wird.

6.4.11.7 Die `Initialise`-Methode

DataHUB ruft diese Methode zu Beginn eines Baus auf, bevor irgendwelche Schichtdaten an das Plug-in gesendet werden. Sie liefert dem Plug-in folglich *bauinvariante Daten*. Das Plug-in kann diese Methode verwenden, um Ressourcen zuzuweisen, die während der Lebensdauer der Instanz zur Berechnung der Baukonstanten usw. benötigt werden.

Parameter: `string`

Ein String, der ein JSON-Objekt enthält, das dem folgenden Schema entspricht:

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

`AssemblyFolder`: `string`

Der Pfad zu dem Ordner mit der Assembly, die dieses Plug-in enthält:

„C:\ProgramData\Renishaw\DataHUB\PlugIns\PlugIn.Example“

`OutputFolder`: `string`

Der Pfad zu dem Ordner, in den die Plug-in-Instanz alle Ausgabedateien schreiben soll. Für jede Plug-in-Instanz, die für einen Auftrag ausgeführt wird, wird ein eigener Ordner im Ausgabeordner des Auftrags erstellt. Der Name dieses Ordners wird aus dem Anzeigenamen und der API-Version des Plug-ins sowie einem Zeitstempel und einer GUID gebildet. Dabei wird die folgende Struktur verwendet:

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

`NumberOfLasers`: `unsigned integer`

Dies ist ein Wert von 1 bis 4, der durch die Hardwarekonfiguration des Rechners, der die Daten erstellt, bestimmt wird.

InitialisationData: JSON object

Der Inhalt, der im Schritt **Plug-in Initialisation data** (Plug-in-Initialisierungsdaten) bei der Auftragserstellung im DataHUB Generator bereitgestellt wird. Weitere Informationen zum Aufbau und zur Verwendung dieses Strings finden Sie unter „Der Initialisierungsstring“ auf Seite 6-97.

Rückgabe: string

Ein String, der ein JSON-Objekt enthält, das dem folgenden Schema entspricht:

```
{
   “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
         [Eine beliebige Anzahl weiterer benutzerspezifischer Eigenschaften]
       }
    ]
 }
```

result: string

Gibt an, ob das Plug-in erfolgreich initialisiert wurde oder nicht. Zurückgegeben werden muss entweder:

- der Literalwert „Success“, der anzeigt, dass das Plug-in erfolgreich initialisiert wurde und mit der Verarbeitung beginnen kann, oder
- der Literalwert „Failure“, der anzeigt, dass das Plug-in auf einen Fehler gestoßen ist und beendet wird, ohne Daten zu empfangen.

Diese Eigenschaft ist obligatorisch.

message: String

Eine Nachricht mit weiteren Details, die an das Ergebnis angehängt und im Audit-Protokoll aufgezeichnet wird. Diese Eigenschaft ist optional. Ist sie nicht vorhanden, wird nur das Ergebnis im Audit-Protokoll aufgezeichnet.

timeSeries: JSON Array

Ein Array, das die Definitionen der Zeitreihen enthält, in denen das Plug-in während der Verarbeitung Daten veröffentlichen wird. Diese Eigenschaft ist optional. Ist sie nicht vorhanden oder ein leeres Array, wird DataHUB keine Zeitreihen in Renishaw Central erstellen.

timeSeries[n].name: String

Name der Zeitreihe. Diese Eigenschaft ist obligatorisch.

timeSeries[n].unitType: String

Die Familie der Einheiten, die diese Daten darstellen. Dies ermöglicht eine nachgelagerte Konvertierung in einen bevorzugten Einheitentyp für die Anzeige. Vordefiniert sind folgende Einheitentypen:

- Acceleration (Beschleunigung)
- Angle (Winkel)
- Binary (Binär)
- Distance (Distanz)
- Flowrate (Flussrate)
- Humidity (Feuchtigkeit)
- MicroDistance (Mikroabstand)
- Percentage (Prozentsatz)
- Pressure (Druck)
- Speed (Geschwindigkeit)
- Temperature (Temperatur)
- Dimensionless (Ohne Dimensionen)

Diese Eigenschaft ist obligatorisch.

timeSeries[n].unit: string

Renishaw Central bevorzugt SI-Einheiten. Es erwartet, dass „pro“-Beziehungen mit dem Zeichen „/“ denotiert werden, und formatiert Potenzen als Zahl. Zum Beispiel wird die Beschleunigung normalerweise in m/s^2 angegeben. Für Temperatur und Grad verwenden Sie gegebenenfalls das Zeichen „°“, zum Beispiel °C für Celsius. Wenn die Werte keine Dimensionen haben, verwenden Sie den Literalwert `Dimensionless`. Diese Eigenschaft ist obligatorisch.

timeSeries[n].absoluteOrRelative: string

Ein Hinweis für Anwendungen, ob die Werte absolut oder relativ zum vorherigen Wert sind. Zurückgegeben werden muss entweder:

- der Literalwert für „Absolute“, der angibt, dass die Werte absolut sind, oder
- der Literalwert für „Relative“, der angibt, dass die Werte in Bezug zum vorhergehenden Wert stehen.

Diese Eigenschaft ist optional. Ist sie nicht vorhanden, wird „Absolute“ verwendet.

timeSeries[n].component: string

Ordnet die Zeitreihe einem bestimmten Aspekt eines Geräts zu, in der Regel einer physischen Einheit wie einer Wetterstation, einem Werkzeug oder einem Softwaresystem. Diese Eigenschaft ist optional. Ist sie nicht vorhanden, wird der Wert „Machine” verwendet.

`timeSeries[n].description:` `string`

Eine Beschreibung der Zeitreihe. Diese Eigenschaft ist optional.

`timeSeries[n].displayHintPrecision:` `unsigned integer`

Ein Hinweis für Anwendungen, die mit Renishaw Central kompatibel sind, mit welcher Genauigkeit die Werte in der Zeitreihe angezeigt werden sollen. Diese Eigenschaft ist optional.

`timeSeries[n].displayHintSampleRate:` `string`

Ein Hinweis für Anwendungen, die mit Renishaw Central kompatibel sind, welche Abtastrate (Sample-Rate) zu verwenden ist. Wird dieser Hinweis weggelassen, nehmen die meisten nachgelagerten Anwendungen an, dass die Standardeinstellung **default** gilt. Verbreitete Sample-Raten sind:

- Raw (Roh)
- Default (Standard)
- Hour (Stündlich)
- Day (Täglich)
- Week (Wöchentlich)
- Month (Monatlich)

Diese Eigenschaft ist optional.

`timeSeries[n].graphType:` `string`

Ein Hinweis für mit Renishaw Central kompatibele Anwendungen, wie die Zeitreihe grafisch dargestellt werden soll. Verbreitete Darstellungsarten sind

- Bar (Balkendiagramm)
- Binary (Binär – insbesondere zur Anzeige von Ein-Aus-Daten)
- Line (Liniendiagramm)

Diese Eigenschaft ist optional.

`timeSeries[n].grouping:` `string`

Die Gruppe, mit der diese Zeitreihe erfasst werden soll. Diese Eigenschaft ist optional.

`timeSeries[n].precision:` `unsigned integer`

Die Genauigkeit, mit der die Daten erfasst/berechnet wurden. Diese Eigenschaft ist optional.

`timeSeries[n]` *Zusätzliche Eigenschaften*

Das Plug-in kann eine beliebige Anzahl zusätzlicher Eigenschaften enthalten, die in Renishaw Central gespeichert werden und für kundenspezifische Software zugänglich sind. Die reservierten Namen „machine“ oder „source“ können hier nicht verwendet werden.

Entspricht der zurückgegebene String nicht dem obigen Schema, wird dies als Fehler behandelt und der vollständige String im Audit-Protokoll erfasst.

6.4.11.8 Die Process-Methode

DataHUB übergibt Teilbereiche der erstellten Daten in aufeinanderfolgenden Aufrufen; die Gesamtzahl der Aufrufe hängt von der Größe der Daten ab. Beim Ausführen dieser Methode leistet das Plug-in den größten Teil seiner Analysearbeit.

Parameter: IReadOnlyCollection<Token>

Ein Abschnitt des Token-Streams für einen Bau.

Rückgabe: `string`

Ausgegeben wird hier ein leerer String oder ein String mit der Meldung „Success“ oder „Failure“, oder ein String, der ein JSON-Objekt enthält, das dem folgenden Schema entspricht:

```
{
   “result”:  string
   “message”: string
   “alerts”:
   [
      {
         “description”: string
         “level”:    string
         “name”:    string
         “time”:    unsigned integer
      }
   ],
   “analysisResults”:
   [
      {
         “name”: string
         “time”: unsigned integer
         Eine beliebige Anzahl weiterer benutzerspezifischer Eigenschaften
      }
   ],
   “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

Gibt an, ob das Plug-in die Daten erfolgreich verarbeiten konnte. Zurückgegeben werden muss entweder ...

- der Literalwert „Success“, der anzeigt, dass das Plug-in die Verarbeitung erfolgreich hat, oder
- der Literalwert „Failure“, der anzeigt, dass das Plug-in auf einen Fehler gestoßen ist. In diesem Fall bleibt das Plug-in aktiv und empfängt weiterhin nachfolgende Daten.

Diese Eigenschaft ist obligatorisch.

message: String

Eine Nachricht mit weiteren Details, die an das Ergebnis angehängt und im Audit-Protokoll aufgezeichnet wird. Diese Eigenschaft ist optional. Ist sie nicht vorhanden, wird nichts im Audit-Protokoll aufgezeichnet.

alerts: JSON Array

Ein Array, das die Meldungen enthält, die das Plug-in während der Verarbeitung an Renishaw Central leiten möchte. Diese Eigenschaft ist optional. Ist sie nicht vorhanden oder ein leeres Array, versucht DataHUB nicht, Meldungen an Renishaw Central zu senden.

alerts[n].description: string

Eine detaillierte Beschreibung der Meldung. Diese Eigenschaft ist obligatorisch.

alerts[n].level: string

Einer der folgenden Literalwerte, der den Schweregrad der Meldung angibt:

- Info
- Warning (Warnung)
- Error (Fehler)

Diese Eigenschaft ist obligatorisch.

alerts[n].name: string

Name der Meldung. Diese Eigenschaft ist obligatorisch.

alerts[n].time: unsigned integer

Der Zeitpunkt, zu dem die Meldung aufgetreten ist, in Mikrosekunden in Relation zur Startzeit der Schicht. Diese Eigenschaft ist obligatorisch Diese Eigenschaft ist obligatorisch.

analysisResults: JSON Array

Ein Array, das die Analyseergebnisse enthält, die das Plug-in während der Verarbeitung an Renishaw Central leiten möchte. Diese Eigenschaft ist optional. Ist sie nicht vorhanden oder ein leeres Array, versucht DataHUB nicht, Analyseergebnisse an Renishaw Central zu senden.

analysisResults [n].name: string

Name der Analyseergebnisse. Diese Eigenschaft ist obligatorisch.

analysisResults [n].time: unsigned integer

Der Zeitpunkt, zu dem das Analyseergebnis erhalten wurde, in Mikrosekunden in Relation zur Startzeit der Schicht. Diese Eigenschaft ist obligatorisch.

analysisResults[n] *Additional Properties*

Das Analyseergebnis kann eine beliebige Anzahl zusätzlicher Eigenschaften enthalten, die in Renishaw Central gespeichert werden und für kundenspezifische Software zugänglich sind. Die folgenden, reservierten Namen dürfen jedoch nicht verwendet werden:

- created (erzeugt)
- machine (Maschine)
- source (Quelle)
- type (Typ)

timeSeries: JSON Array

Ein Array mit Zeitreihendaten, die das Plug-in zu den Zeitreihen hinzufügen möchte, die in der *Initialise*-Methode definiert sind. Diese Eigenschaft ist optional. Ist sie nicht vorhanden oder ein leeres Array, wird DataHUB die Daten der Zeitreihen nicht in Renishaw Central aktualisieren.

timeSeries[n].name: String

Der Name der Zeitreihe wie bei der Initialisierung definiert. Diese Eigenschaft ist obligatorisch.

timeSeries[n].unitType: String

Der Typ der Zeitreihe wie bei der Initialisierung definiert. Diese Eigenschaft ist obligatorisch.

timeSeries[n].unit: string

Die Einheit der Zeitreihe wie bei der Initialisierung definiert. Diese Eigenschaft ist obligatorisch.

timeSeries[n].values: JSON Array

Ein Array mit den Zeit- und Werte-Paaren, die der Zeitreihe hinzugefügt werden sollen. Eine Zeitreihe muss entweder die Eigenschaft „values“ oder „limits“ oder beide aufweisen.

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

Die zugehörige Zeit für den Wert, in µs in Relation zur Startzeit der Schicht. Diese Eigenschaft ist obligatorisch.

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

Der Wert. Diese Eigenschaft ist obligatorisch.

timeSeries[n].limits: JSON Array

Ein Array, das die Grenzwerte für die Daten enthält. Es gibt drei verschiedene Arten von Grenzwerten:

1. Operating (Betrieb)
2. Warning (Warnung)
3. Display (Anzeige)

Anwendungen, die mit Renishaw Central kompatibel sind, können diese Grenzwerte nutzen, um die Anzeige und das Auslösen von Aktionen zu unterstützen, wenn Zeitseriendatenpunkte die jeweiligen Grenzwerte überschreiten.

Die Grenzwerte gelten ab dem in der Eigenschaft „time“ vorgegebenen Zeitpunkt, bis zu dem vom nächsten hinzugefügten Grenzwert vorgegebenen Zeitpunkt. Die zuletzt festgelegten Grenzwerte gelten unbefristet.

Es ist nicht erforderlich, dass ein Plug-in Grenzwerte für eine Zeitreihe bereitstellt, denn es ist durchaus möglich, dass die Daten keinen natürlichen Betriebsbereich oder sinnvollen Anzeigebereich haben. Es ist auch nicht notwendig, dass sich einmal festgelegte Grenzwerte jemals ändern. Die Möglichkeit, sie zu ändern, ist für Fälle vorgesehen, in denen eine Änderung angebracht wäre.

Eine Zeitreihe muss entweder die Eigenschaft „values“ oder „limits“ oder beide aufweisen.

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

Der Zeitpunkt, ab dem ein Grenzwert gelten soll, in µs in Relation zur Startzeit der Schicht. Diese Eigenschaft ist obligatorisch.

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

Der untere Anzeigegrenzwert. Diese Eigenschaft ist optional. Wenn sie nicht vorhanden ist, gilt weiterhin der aktuelle untere Anzeigegrenzwert.

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

Der obere Anzeigegrenzwert. Diese Eigenschaft ist optional. Wenn sie nicht vorhanden ist, gilt weiterhin der aktuelle obere Anzeigegrenzwert.

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

Der untere Grenzwert für eine Warnmeldung. Diese Eigenschaft ist optional. Wenn sie nicht vorhanden ist, gilt weiterhin der aktuelle untere Meldegrenzwert.

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

Der obere Grenzwert für eine Warnmeldung. Diese Eigenschaft ist optional. Wenn sie nicht vorhanden ist, gilt weiterhin der aktuelle obere Meldegrenzwert.

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

Der untere Betriebsgrenzwert. Diese Eigenschaft ist optional. Wenn sie nicht vorhanden ist, gilt weiterhin der aktuelle untere Betriebsgrenzwert.

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

Der obere Betriebsgrenzwert. Diese Eigenschaft ist optional. Wenn sie nicht vorhanden ist, gilt weiterhin der aktuelle obere Betriebsgrenzwert.

Entspricht der zurückgegebene String nicht dem obigen Schema, wird dies als Fehler behandelt und der vollständige String im Audit-Protokoll erfasst.

6.4.11.9 Die *Shutdown*-Methode

Die *Shutdown*-Methode wird nur einmal aufgerufen und zwar dann, wenn entweder die *Initialise*-Methode einen Fehler zurückgibt oder wenn keine weiteren Daten zu verarbeiten sind (weil alle erwarteten Daten verarbeitet wurden oder der Auftrag abgebrochen wurde). Nach diesem Aufruf erhält das Plug-in keine weiteren Aufrufe.

Diese Methode signalisiert, dass das Plug-in sämtliche abschließenden Verarbeitungen an den Baudaten vornehmen, alle eventuell übernommenen Ressourcen freigeben soll usw.

Parameter:

Keine.

Rückgabe: `string`

Wenn der String das Wort „Success“ enthält, wurde das Plug-in erfolgreich beendet. Andernfalls bedeutet dies, dass beim Plug-in ein Fehler aufgetreten ist. In beiden Fällen wird der String vollständig in den Audit-Protokollen aufgezeichnet.

6.4.11.10 Der Initialisierungsstring

DataHUB initialisiert V2.0-Plug-ins durch einen String im JSON-Format. Dieser String enthält die Werte für die globalen Bauparameter sowie den Plug-in-Initialisierungsstring, der im DataHUB Generator-Workflow für die Verarbeitung des Baus durch das Plug-in angegeben ist.

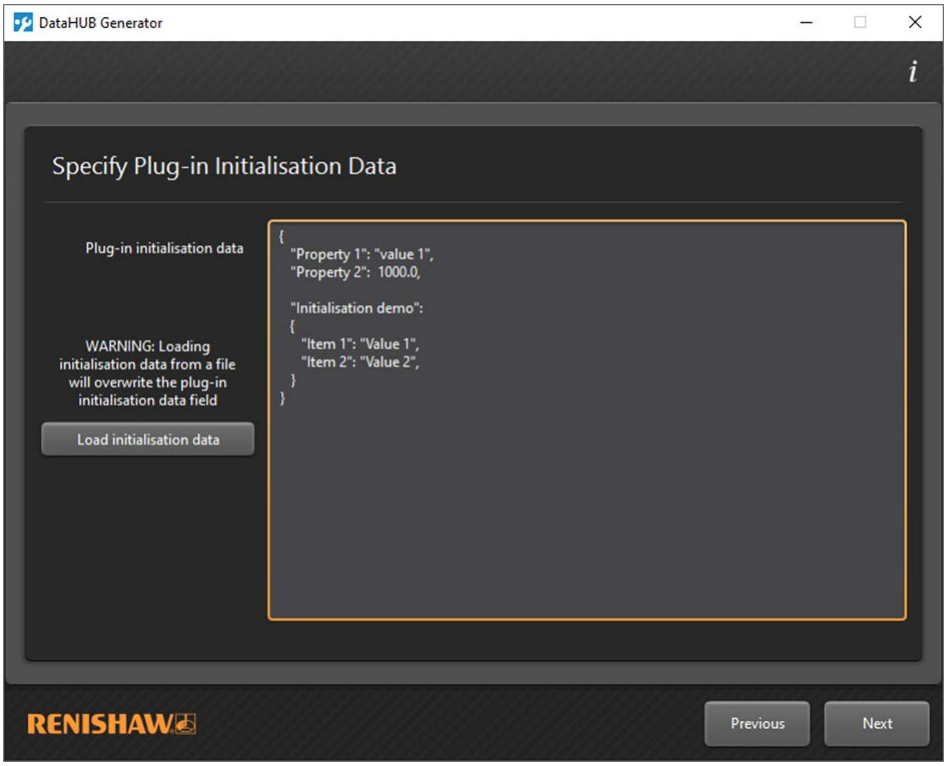


Abbildung 42 JSON-String für die Plug-in-Initialisierung

Der in den DataHUB Generator-Workflow eingegebene String ist ebenfalls JSON-formatiert. Sie können Eigenschaften und Objekte hinzufügen, um für jeden Plug-in-Typ spezifische Daten an die Initialisierungsmethode der Instanz zu senden.

Der Initialisierungsstring für das Plug-in könnte beispielsweise so aussehen:

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

Die Verwendung einer JSON-Parser-Bibliothek (z. B. Newtonsoft.Json oder System.Text.Json) im Plug-in vereinfacht das Extrahieren der im Initialisierungsstring enthaltenen Werte. Zum Beispiel:

```
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”];
```

Die im DataHUB Generator angegebenen Instanz-Initialisierungsdaten sind ebenfalls ein JSON-String. Beachten Sie also vor dem Zugriff auf die enthaltenen Eigenschaften, dass sie als JObject geparst sind. Die Werte der Eigenschaften `Property1` und `Property2` sowie die Eigenschaftswerte im verschachtelten Demoobjekt `Initialisation` werden im VisualStudio-Überwachungsfenster angezeigt:

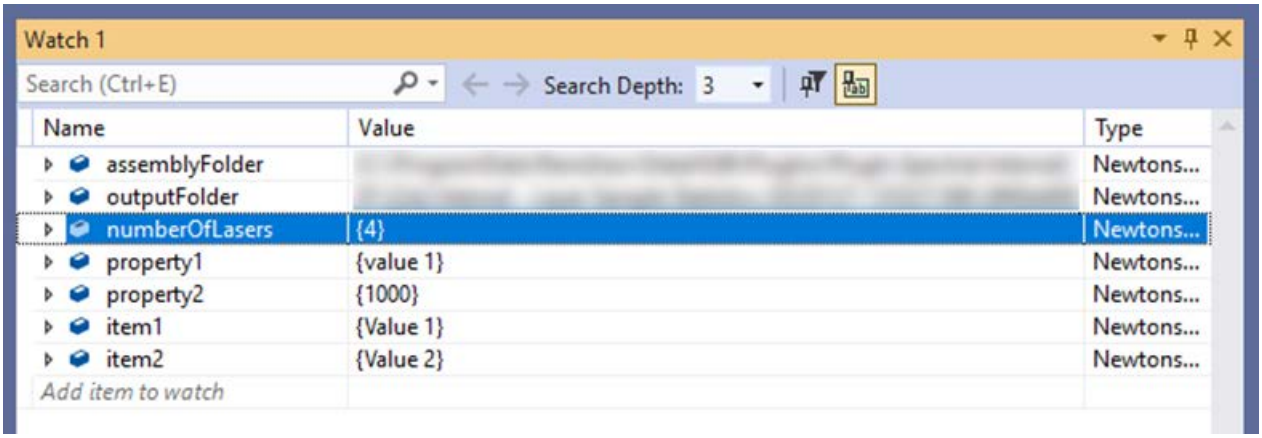


Abbildung 43 Plug-in-Initialisierungsdatenstring beim Aufruf von „Initialise“

Da auf einem DCPC in der Regel mehrere Plug-ins verwendet werden, empfiehlt es sich, zur Formatierung des DataHUB Generator-Strings für jeden Plug-in-Typ ein benanntes JSON-Objekt anzugeben:

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

Der Zugriff auf die spezifischen Initialisierungsdaten für das Plug-in bedeutet folglich den Zugriff auf das zugehörige verschachtelte Objekt, d. h. in Plug-in TypeA bzw. TypeB:

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

6.4.11.11 Token-Typen

6.4.11.11.1 DataSourceInformation

Dieses Token beschreibt die Eigenschaften einer Datenquelle für die Prozessüberwachung.

```
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

Dieses Token markiert das Ende aller Daten einer Schicht.

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

6.4.11.11.3 EndOfLayerLaserData

Dieser Token markiert das Ende der Samples für einen Laser und eine Schicht.

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

6.4.11.11.4 PacketOfSamples

Dieses Token enthält eine Sammlung von Samples.

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

6.4.11.11.5 Sample

Ein Sample-Token enthält Daten, die zu einem bestimmten Zeitpunkt während des Schichtbaus gesammelt wurden. Pro Schicht und Laser wird es also viele Sample-Tokens geben.

- `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

Dieses Token hat keine Eigenschaften.

6.4.11.11.7 StartOfLayerData

Dieses Token markiert den Beginn der Token-Reihe für eine Schicht.

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

6.4.11.11.8 StartOfLayerLaserData

Dieses Token markiert den Beginn der Token-Reihe für einen Laser und eine Schicht.

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

6.4.11.11.9 SummaryPacket

Dieses Token enthält durch die Summary-Methoden erzeugte Werte, die für den Filter `SummarisePacketOfSamples` bereitgestellt werden. Ein Satz dieser Summary-Werte ist in Instanzen der verschachtelten Klasse `SummaryValues` enthalten. Da Summary-Methoden die Ergebnisse als Bruchwert ausgeben können, hat jede Eigenschaft in `SummaryValues` den Typ „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

Die Klasse, die allen Token zugrunde liegt. Diese Klasse enthält keine Daten.

6.4.11.11.11 LaserID und LayerNumber

Laserkennungen und Schichtnummern sind generell ganzzahlig. Dies sorgt für Typsicherheit bei der Konstruktion der verschiedenen obengenannten Tokens, beispielsweise um versehentliche Fehlzuweisungen der Schichtnummer zur Laserkennung zu verhindern.

6.4.11.12 Filter

Die API bietet eine Reihe von Filtern für allgemeine Aufgaben bei der Verarbeitung eines Token-Streams.

6.4.11.12.1 EmitSampleOnlyIf

Dieser Filter vergleicht den Wert einer der Eigenschaften eines eingegebenen `Sample`-Tokens und gibt das `Sample` nur dann aus, wenn der Wert als korrekt bestätigt wurde. Alle Token-Typen außer `Sample` werden in den Ausgabe-Stream abgegeben.

Die folgenden Eigenschaftswerte sind verfügbar:

- DemandX (angeforderte X-Position)
- DemandY (angeforderte Y-Position)
- DemandFocus (angeforderter Fokus)
- DemandLaserPower (angeforderte Laserleistung)
- MeltVIEWPlasma
- MeltVIEWMeltpool
- LaserVIEW
- LaserOutputPower (Laser-Ausgabeleistung)
- LaserBackReflection (Laserrückreflexion)
- LaserModulate (Lasermodulation)
- Die Gleichheitsprüfungen sind:
- EqualTo (gleich)
- NotEqualTo (nicht gleich)
- GreaterThan (größer als)
- GreaterThanOrEqualTo (größer als oder gleich)
- LessThan (kleiner als)
- LessThanOrEqualTo (kleiner als oder gleich)
- CustomTest (benutzerspezifische Prüfung)
- IsOn („Ist EIN“ – nur für Lasermodulation)

Mit der letzteren Variante des Filters kann ein anpassbarer Vergleichstest konfiguriert werden. Jede Funktion, die zwei ganzzahlige Eingaben annimmt und einen booleschen Wert zurückgibt, kann verwendet werden. Zum Beispiel:

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

Hier würde eine benutzerdefinierte Prüfung Samples mit einem geraden DemandX-Wert akzeptieren.

6.4.11.12.2 SplitWhen

Dieser Filter vergleicht den Eigenschaftswert eines `Sample`-Eingabetokens mit dem Eigenschaftswert des vorangehenden Tokens und fügt bei Abweichungen vor der Ausgabe des Eingabetokens ein `Split`-Token ein, um beide Tokens zu trennen. Alle Token-Typen außer `Sample` werden in den Ausgabe-Stream abgegeben.

Die folgenden Varianten stehen zur Verfügung:

- DemandXChanges (Veränderungen der angeforderten X-Position)
- DemandYChanges (Veränderungen der angeforderten Y-Position)
- DemandFocusChanges (Veränderungen des angeforderten Fokus)
- LaserModulateChanges (Veränderungen der Lasermodulation)
- DemandLaserPowerChanges (Veränderungen der angeforderten Laserleistung)

HINWEIS: Nur erforderte Eigenschaften werden hier bereitgestellt; die anderen Prozessüberwachungs-eigenschaften variieren stark von Sample zu Sample und sind ungeeignet zur Aufteilung des Token-Streams.

6.4.11.12.3 CollateInto

Dieser Filter fasst alle von `Split`-Tokens umgebenen `Sample`-Tokens in einem einzigen `PacketOfSamples`-Token zusammen. Die `Split`-Tokens und `Sample`-Tokens werden dabei geschluckt. Alle anderen Token-Typen werden in den Ausgabe-Stream abgegeben.

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

Das Ausmaß dieser Zuordnung ist erwähnenswert. Bei einem Token-Stream, der zuvor durch `Split`-Tokens aufgeteilt wurde (wenn sich beispielsweise die Lasermodulation, Position usw. geändert hatten), enthält der resultierende Ausgabe-Stream keine `Sample`- oder `Split`-Tokens, da diese durch `PacketOfSamples`-Tokens ersetzt wurden.

6.4.11.12.4 SummarisePacketOfSamples

Dieser Filter verarbeitet die von `CollateInto` erzeugten `PacketOfSamples`-Tokens und führt eine oder mehrere Summary-Funktionen für die zusammengefassten Sample-Datenwerte aus. Der Konstruktor dieses Filters erlaubt das Konfigurieren einer oder mehrerer benannter Summary-Funktionen, deren Ergebnisse jeweils im Ausgabe-Token gespeichert werden.

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

Eine SummaryMethod-Instanz wird mit einem Namen und einer Summary-Funktion gebildet:

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

6.4.11.12.5 Pipeline

Für den Zusammenbau von Filtern in Pipelines werden die folgenden beiden Helfer benötigt:

```
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 Helfer

Die API bietet einige Helferklassen zur Definition von allgemeinen Operationen beim Schreiben von Plug-ins.

6.4.11.13.1 Positionshelfer

Diese Klasse bietet Funktionen zum einheitlichen Runden von Double-Werten:

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

6.4.11.13.2 Rückgabestring-Helfer

Die API bietet Helfermethoden zur Erstellung der von DataHUB benötigten Rückgabestrings in jeder Phase des Plug-ins (*Initialise*, *Process* und *Shutdown*).

HINWEIS: DataHUB erwartet einen gültigen JSON-Rückgabestring, welcher der Rückgabestring-API entspricht. Sonderzeichen in Strings müssen mit einem zusätzlichen Backslash versehen werden. Andernfalls könnte ein ungültiger JSON-String an DataHUB gesendet und dort verworfen werden. Zum Beispiel:

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

Ein zweites Beispiel:

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

6.4.11.13.3 Rückgabe aus „Initialise“

Helfer zur Erstellung einfacher Plug-in-Rückgabestrings von „Initialise“ zu DataHUB sind:

```
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);
```

Um eine oder mehrere Zeitreihen in Renishaw Central zu erstellen, können die folgenden Helfer verwendet werden:

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

Ein `TimeSeries`-Objekt muss über einen Namen, einen Einheitentyp und eine Einheit verfügen. Das `TimeSeries`-Objekt „Factory“ bietet eine fließende Schnittstelle mit bekannten Einheitentypen und Einheiten und macht die Konstruktion zu einem einfachen, weniger fehleranfälligen Prozess. Dies wird im nachstehenden Beispiel demonstriert:

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

6.4.11.13.4 Rückgaben aus „Process“

Helfer zur Erstellung einfacher Plug-in-Rückgabestrings aus *Prozess* zurück zu DataHUB sind:

```
  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);
```

Mit den folgenden Helfern können Plug-in-Rückgabestrings aus der Process-Funktion verwendet werden, um Erfolg oder Fehlschlag zu signalisieren und Daten wie Zeitreihenaktualisierungen, Meldungen und Analyseergebnisse an Renishaw Central zu senden.

```
  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`-Zeitreihen, die im Rahmen von *Initialise* definiert wurden, können mit Werten, Grenzwerten oder beidem aktualisiert werden. Wie das nachstehende Beispiel zeigt, bietet das `TimeSeries`-Objekt eine Update-Methode, die ein `UpdateTimeSeries`-Objekt zurückgibt:

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

Um Meldungen an Renishaw Central auszugegeben, kann ein `Alert`-Objekt konstruiert und an die entsprechende `SendData`-Helfermethode übergeben wird. Wie das nachfolgende Beispiel zeigt, lässt sich dies leicht durch `Alert.Factory` bewerkstelligen:

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

Und letztlich können durch Konstruieren eines `AnalysisResult`-Objekts und das nachfolgende Übergeben an die `SendData`-Helfermethode auch Analyseergebnisse an Renishaw Central weitergeleitet werden. Das nachfolgende Beispiel zeigt, wie dies durch `AnalysisResult.Factory` ermöglicht wird:

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

6.4.11.13.5 Rückgabe aus „Shutdown“

Helfer zur Erstellung einfacher Plug-in-Strings aus der *Shutdown*-Funktion zur Rückgabe an DataHUB sind:

```
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 Das ExportPackets Plug-in

Die LaserVIEW- und MeltVIEW-Hardware sammelt Daten, die mit einer Rate von 100 kHz abgetastet werden, wobei jedes einzelne Sample mehrere Werte enthält. Während Daten mit dieser Auflösung für manche Analyseaufgaben nützlich sind, kann es für viele Analyseaufgaben ausreichend sein, die Samples mit einer geringeren Auflösung darzustellen. Das Plug-in fasst daher die Daten in sogenannten Paketen zusammen. Ein Paket muss dabei folgende Voraussetzungen erfüllen:

- Der Laser wurde gezündet und
- die angeforderten X- und Y-Position blieben konstant.

Anhand dieser Definition wird eine Reihe von Messergebnissen (Samples) erfasst, die während eines einzigen Zündens des Lasers auf dem Pulverbett gewonnen wurden. Für jeden Satz an Samples werden sowohl der Mittelwert als auch der Median aus den vom Laser/MeltVIEW-Prozessüberwachungssensor erfassten Werten ermittelt. Es ist zu beachten, dass die Laser/MeltVIEW-Werte der Samples zu Beginn einer Exposition aufgrund der Betriebseigenschaften des Lasers etwas niedriger sein können. Der Medianwert kann daher ein aussagekräftigeres Bild der Sample-Daten liefern als der Mittelwert.

6.5.1 Plug-in ausführen

Um dieses Plug-in zu verwenden, führen Sie den DataHUB Generator aus und wählen Sie den Ordner *Build Data* (Baudaten) aus, der die Dateien mit den Prozessüberwachungsdaten für den Bau enthält. Wählen Sie einen Ausgabeordner für das Plug-in. Falls erforderlich (z. B. wenn die Datei *BuildStarted yyyy.mm.dd.HH.mm.ss.dhxml* mit den bauspezifischen Informationen nicht im Eingabeordner gefunden wurde), stellen Sie sicher, dass unter *Number of Lasers* (Anzahl Laser) die korrekte Anzahl an Laser eingestellt wurde. Das Plug-in kann nur dann Daten verarbeiten, wenn die ausgewählte Anzahl an Lasern mit der tatsächlichen Anzahl an Lasern der mit der LaserVIEW/MeltVIEW-Hardware ausgestatteten AM-Maschine übereinstimmt, auf der die Daten erzeugt wurden. Die beiden anderen Werte, *Layer Spacing* (Schichtabstand) und *Build Height* (Bauhöhe) können unbearbeitet bleiben, da sie vom Plug-in nicht benötigt werden.

Geben Sie eine Beschreibung ein und drücken Sie auf **Next** (Weiter).

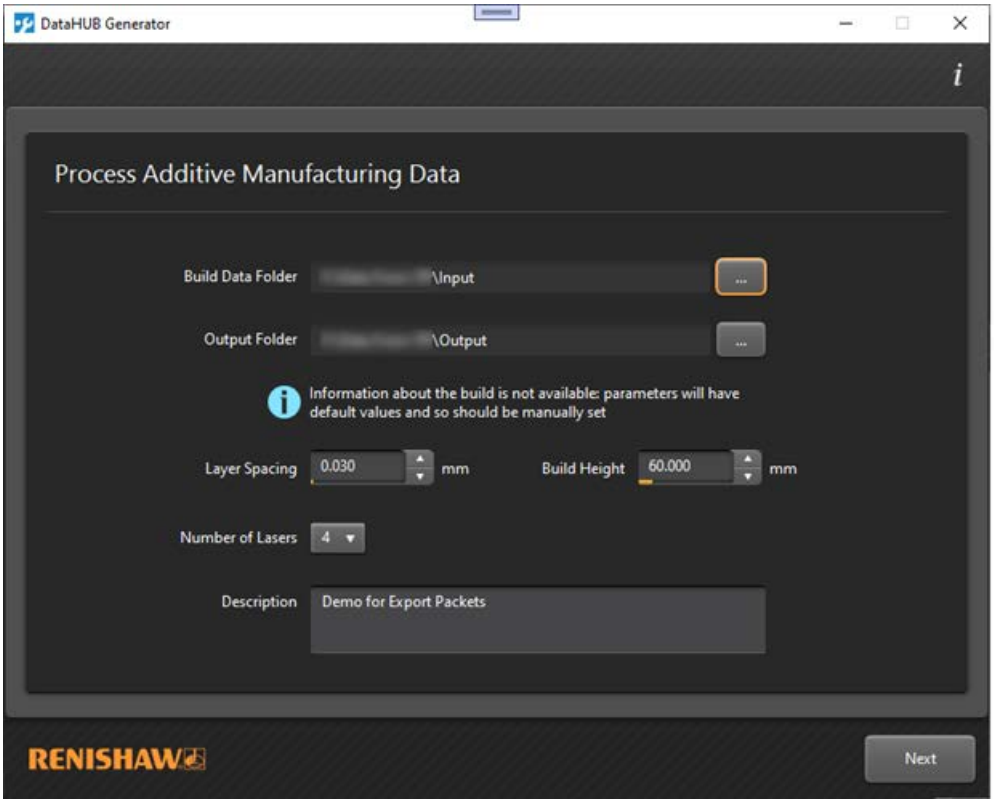


Abbildung 44 Workflow Schritt 1 – Daten konfigurieren

Wählen Sie im nächsten Teil des Arbeitsablaufs **Process LaserVIEW and/or MeltVIEW data via plug-ins** (LaserVIEW- und/oder MeltVIEW-Daten über Plug-ins verarbeiten) und klicken Sie auf **Next** (Weiter).

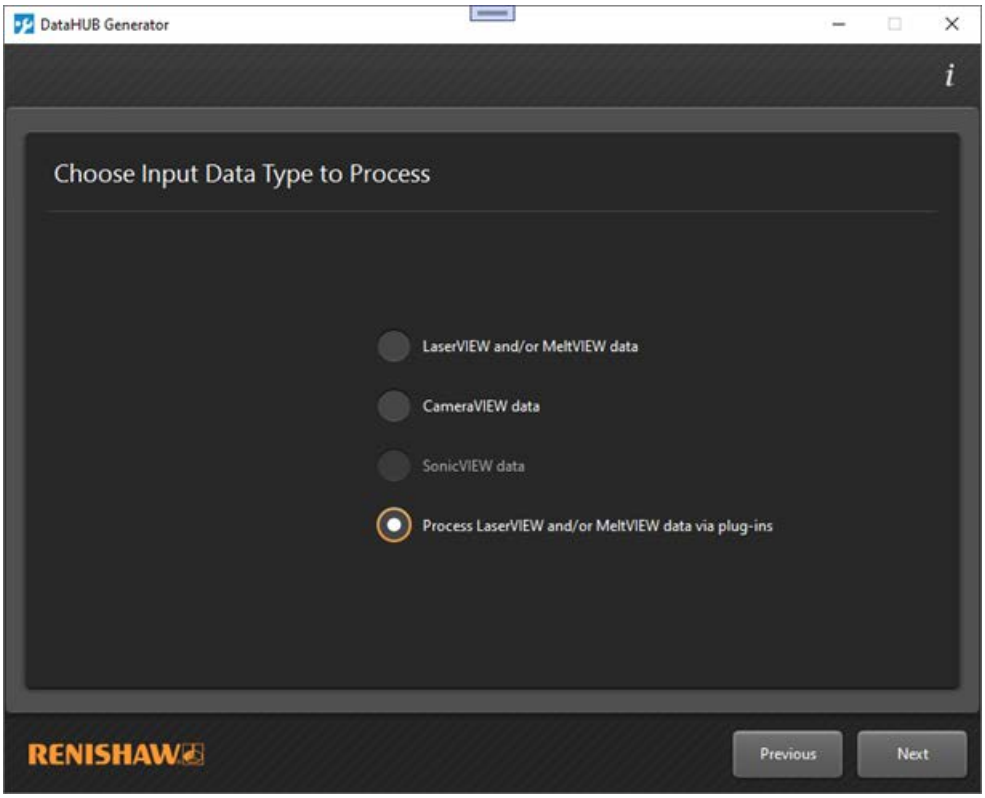


Abbildung 45 Workflow Schritt 2 – Verarbeitung durch das Plug-in auswählen

Da dieses Plug-in keine Initialisierungsdaten benötigt, kann das Textfeld leer gelassen werden. Klicken Sie auf **Next** (Weiter).

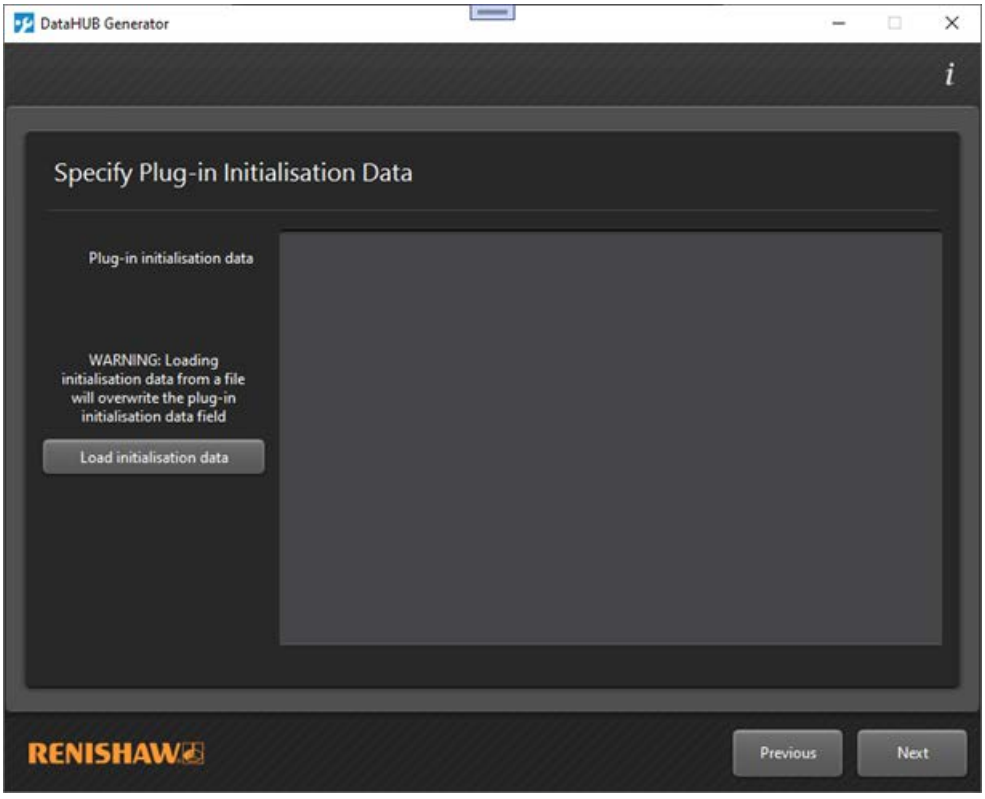


Abbildung 46 Workflow Schritt 3 – Plug-in initialisieren

Im letzten Schritt dieses Arbeitsablaufs klicken Sie auf **Trigger LaserVIEW/MeltVIEW Plug-in Processing** (LaserVIEW/MeltVIEW Plug-in Verarbeitung auslösen).

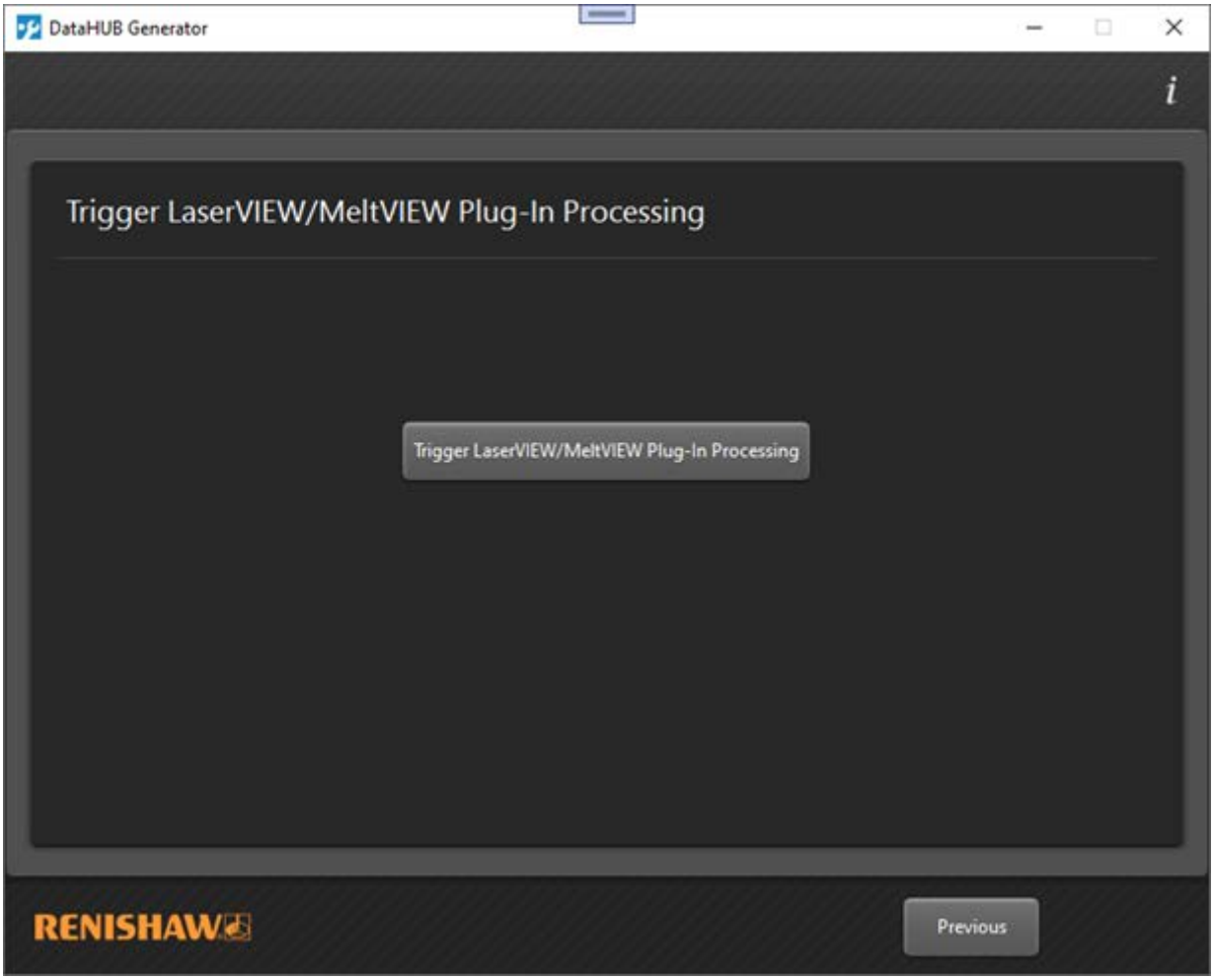


Abbildung 47 Workflow Schritt 4 – Verarbeitung beginnen

Die DataHUB Generator App kann nun beendet werden.

6.5.2 Dateinamen und Speicherort

Der im vorstehenden Workflow unter *Output* angegebene Ausgabeordner ist der Stammordner, in den das Plug-in seine Ausgabe schreibt. Das Plug-in erstellt dort einen Unterordner mit folgendem Namen:

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

Dabei ist **yyyyMMdd-HHmmss.ff** das Datum und die Uhrzeit, zu der das Plug-in mit der Verarbeitung der Daten begonnen hat, und **GUID** ein eindeutiger Code. Diese Namensgebung soll sicherstellen, dass das Plug-in beim erneuten Ausführen mit denselben Daten bereits vorhandene Ausgaben nicht überschreibt.

In den Unterordner schreibt das Plug-in für jeden Laser und jede Schicht eine Datei mit dem Namen **Packet data for layer x, laser y.txt** (Paketdaten für Schicht x, Laser y.txt), wobei x die Nummer der Schicht und y die Nummer des Lasers ist. Beachten Sie, dass die Datei zwar die Dateierweiterung „txt“ verwendet, aber ein einfaches tabulatorgetrenntes Format aufweist, sodass sie ohne Vorverarbeitung in viele gängige Softwarepakete importiert werden kann.

6.5.3 Dateiinhalt

Jede Datei beginnt mit einer Reihe von Spaltenüberschriften:

- Start time (Startzeit): bezogen auf den Beginn der Schicht (µs)
- Duration (Dauer): abgeleitet von der Zeit des ersten und des letzten im Paket erfassten Samples (µs)
- Demand X (Angeforderte X-Position): Position des Pakets auf dem Pulverbett (–125,0 mm links bis +125,0 mm rechts)
- Demand Y (Angeforderte Y-Position): Position des Pakets auf dem Pulverbett (–125,0 mm vorne bis +125,0 mm hinten)
- Demand focus (Angeforderter Fokus): Fokussierung des Laserstrahls auf dem Pulverbett (±5,0 mm)
- Demand laser power (mean) (Durchschnittliche angeforderte Laserleistung): Gemittelte Werte für die im Paket erfassten Samples
- MeltVIEW plasma (mean) (MeltVIEW Plasma Mittelwert): Gemittelte Werte für die im Paket erfassten Samples
- MeltVIEW Melt Pool (mean) (MeltVIEW Schmelzbad Mittelwert): Gemittelte Werte für die im Paket erfassten Samples
- LaserVIEW (mean) (LaserVIEW Mittelwert): Gemittelte Werte für die im Paket erfassten Samples
- Laser back reflection (mean) (Laserrückreflexion Mittelwerte): Gemittelte Werte für die im Paket erfassten Samples
- Laser output power (mean) (Durchschnittliche Laserleistung): Gemittelte Werte für die im Paket erfassten Samples
- Demand laser power (median) (Mediane angeforderte Laserleistung): Medianwert für die im Paket erfassten Samples
- MeltVIEW plasma (median) (MeltVIEW Plasma Median): Medianwert für die im Paket erfassten Samples
- MeltVIEW Melt Pool (median) (MeltVIEW Schmelzbad Median): Medianwert für die im Paket erfassten Samples
- LaserVIEW (median) (LaserVIEW Median): Medianwert für die im Paket erfassten Samples
- Laser back reflection (median) (Laserrückreflexion Median): Medianwert für die im Paket erfassten Samples
- Laser output power (median) (Mediane Laserleistung): Medianwert für die im Paket erfassten Samples

Darauf folgt eine Anzeige der zusammengefassten Werte für jedes Paket an AMPM-Daten:

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

Leere Seite

RENISHAW
apply innovation™

www.renishaw.de/Renishaw-Weltweit

#renishaw

Renishaw GmbH
T +49 (0)7127 9810
E germany@renishaw.com

Renishaw (Austria) GmbH
T +43 2236 379790
E austria@renishaw.com

Renishaw (Switzerland) AG
T +41 55 415 50 60
E switzerland@renishaw.com

Artikel-Nr.: H-5800-6856-01-A
Veröffentlicht: 10.2023