SERENEDI

HEALTHCARE INTEGRATION PLATFORM

ORACLE CLOUD, RELEASE 20201115

Contents

Licensing

Usage of SERENEDI is governed by the licensing terms accepted by the user when provisioning the SERENEDI Instance.

The software is licensed for use only to the User's employees or contractors. Chiapas EDI Technologies, Inc. disclaims any warranties, express or implied, about the suitability or functionality of this software for a particular purpose.

The Electronic Data Interchange standards used in this software are copyrighted by the Accredited Standards Committee X12 (ASC X12). Used under license.

SERENEDI uses the following libraries:

Under the MIT License:

Blazorise, Copyright © 2020 Mladen Macanovic

SSH.NET, Copyright © 2016 Renci

WixSharp, Copyright © 2016 Oleg Shilo

Copyright © 2020 Microsoft:

.NET Core 3.1
Microsoft.Management.Infrastructure
PowerShellStandard.Library
System.Data.SqlClient
Microsoft.PowerShell.SDK
Microsoft.WSMan.Management
Microsoft.PowerShell.Commands.Diagnostics

Under the Apache 2.0 license:

BlazorInputFile, Copyright © 2020 Steve Sanderson

Oracle Managed Data Access Client Terms

Oracle Managed Data Access Core Driver, Copyright © Oracle, 2019.

The following terms and conditions apply only to the embedded ODP.NET driver supplied by Oracle that provides Oracle database access to SERENEDI. Oracle is a Third-Party beneficiary to this agreement.

License Rights and Restrictions

Oracle grants You a nonexclusive, nontransferable, limited license to, subject to the restrictions stated in this Agreement, (a) internally use the ODP.NET Drivers solely for the purposes of developing, testing, prototyping and demonstrating Your applications, and running the ODP.NET Drivers for Your own internal business operations. You may allow Your Contractor(s) to use the ODP.NET Drivers, provided they are acting on Your behalf to exercise license rights granted in this Agreement and further provided that You are responsible for their compliance with this Agreement in such use. You will have a written agreement with Your Contractor(s) that strictly limits their right to use the ODP.NET Drivers and that otherwise protects Oracle’s intellectual property rights to the same extent as this Agreement. You may make copies of the ODP.NET Drivers to the extent reasonably necessary to exercise the license rights granted in this Agreement. You may make one copy of the ODP.NET Drivers for backup purposes.

Further, You may not:

- Distribute the ODP.NET Driver and it’s Documentation to third parties;

- remove or modify any ODP.NET Driver markings or any notice of Oracle's or a licensor's proprietary rights;
- use the ODP.NET Drivers to provide third party training unless Oracle expressly authorizes such use on the ODP.NET Driver's download page;
- cause or permit reverse engineering (unless required by law for interoperability), disassembly or decompilation of the ODP.NET Drivers; and
- disclose results of any ODP.NET Driver benchmark tests without Oracle's prior consent.

The ODP.NET Drivers may contain source code that, unless expressly licensed in this Agreement for other purposes (for example, licensed under an open source license), is provided solely for reference purposes pursuant to the terms of this Agreement and may not be modified.

All rights not expressly granted in this Agreement are reserved by Oracle. If You want to use the ODP.NET Drivers or Your application for any purpose other than as expressly permitted under this Agreement, You must obtain from Oracle or an Oracle reseller a valid ODP.NET Drivers license under a separate agreement permitting such use. However, You acknowledge that the ODP.NET Drivers may not be intended for production use and/or Oracle may not make a version of the ODP.NET Drivers available for production or other purposes; any development or other work You undertake with the ODP.NET Drivers is at Your sole risk.

Ownership

Oracle or its licensors retain all ownership and intellectual property rights to the ODP.NET Drivers.

Export Controls

Export laws and regulations of the United States and any other relevant local export laws and regulations apply to the ODP.NET Drivers . You agree that such export control laws govern Your use of the ODP.NET Drivers (including technical data) and any services deliverables provided under this agreement, and You agree to comply with all such export laws and regulations (including "deemed export" and "deemed re-export" regulations). You agree that no data, information, ODP.NET Driver and/or materials resulting from ODP.NET Drivers or services (or direct products thereof) will be exported, directly or indirectly, in violation of these laws, or will be used for any purpose prohibited by these laws including, without limitation, nuclear, chemical, or biological weapons proliferation, or development of missile technology. Accordingly, You confirm:

- You will not download, provide, make available or otherwise export or re-export the ODP.NET Drivers, directly or indirectly, to countries prohibited by applicable laws and regulations nor to citizens, nationals or residents of those countries.
- You are not listed on the United States Department of Treasury lists of Specially Designated Nationals and Blocked Persons, Specially Designated Terrorists, and Specially Designated Narcotic Traffickers, nor are You listed on the United States Department of Commerce Table of Denial Orders.
- You will not download or otherwise export or re-export the ODP.NET Drivers, directly or indirectly, to persons on the above mentioned lists.
- You will not use the ODP.NET Drivers for, and will not allow the ODP.NET Drivers to be used for, any purposes prohibited by applicable law, including, without limitation, for the development, design, manufacture or production of nuclear, chemical or biological weapons of mass destruction.

Disclaimer of Warranties; Limitation of Liability

THE ODP.NET DRIVERS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ORACLE FURTHER DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT .

IN NO EVENT WILL ORACLE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF ORACLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLE'S ENTIRE LIABILITY FOR DAMAGES UNDER THIS AGREEMENT SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000) .

No Technical Support

- Unless Oracle support for the ODP.NET Drivers, if any, is expressly included in a separate, current support agreement between You and Oracle, Oracle's technical support organization will not provide technical support, phone support, or updates to You for the ODP.NET Drivers provided under this Agreement.

Termination

- You may terminate this Agreement by destroying all copies of the ODP.NET Drivers. This Agreement shall automatically terminate without notice if You fail to comply with any of the terms of this Agreement, in which case You shall promptly destroy all copies of the ODP.NET Drivers.

Relationship Between the Parties

- Oracle is an independent contractor and we agree that no partnership, joint venture, or agency relationship exists between us. We each will be responsible for paying our own employees, including employment related taxes and insurance. Nothing in this agreement shall be construed to limit either party's right to independently develop or distribute software that is functionally similar to the other party's products, so long as proprietary information of the other party is not included in such software.

U.S. Government End Users

- ODP.NET Drivers and/or ODP.NET Drivers Documentation delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the ODP.NET Drivers and/or ODP.NET Drivers Documentation shall be subject to the license terms and license restrictions set forth in this Agreement. No other rights are granted to the U.S. Government.

Introduction

Integration is the combining of two systems so they are synchronized to act as one. Healthcare integration is the combining and synchronizing of different healthcare systems, often across corporate boundaries, so they act as one. A simple example of this is *eligibility*. If a provider group has a three-month-old snapshot of eligibility from the payer, it could be sending claims to the payer for a patient who was disenrolled two months ago. These claims would be denied by the payer, and the provider would be in the position of trying to collect payment from the patient. If both the payer's and provider's eligibility systems were tightly integrated, this problem would cease to exist.

In the "old days" of healthcare, this integration was generally handled with flat files or CSV (Comma Separated Values) files that were customized between trading partners as needed. In some cases, very large trading partners, like Medicare, generated large, complex flat files that were consumed universally across the country. Although there were some attempts to universalize file formats, little to no legislation supported them and therefore they lacked incentive. As a result, the late 1990s saw the healthcare industry with millions of proprietary data formats, and even small changes in business requirements demanded heavy investment in development time.

Enter the Healthcare Insurance Portability Administration Act, or HIPAA: by federal law, trading partners would no longer be allowed to exchange variable and custom file formats. Instead, the ASC X12 committee agreed on a fixed file format for

a specific business-to-business transaction and authored something called a *HIPAA Implementation Guide*, and then this guide became the final law to determine what the file would look like and what data it could contain.

These implementation guides established a Rosetta Stone for the healthcare industry. The authors of the guides used their industry knowledge to encapsulate all the different situations that can occur during these common business functions, such as transmitting eligibility from a payer to a provider, and created a file format that could accommodate many different business needs.

These new hierarchical data formats vastly differ from the CSV or flat files that came before. Furthermore, many of the tools and technologies available to tackle these complex, hierarchical formats have a steep learning curve and a long development time. SERENEDI is a third-generation healthcare integration platform—building on the 2003 product Chiapas Version 1, and the 2012 product, Chiapas EDI Enterprise—designed specifically to address this complexity.

Designed by and for healthcare IT professionals, SERENEDI was in development for six years before we considered it ready for release, and it brings powerful technology to bear on modern integration challenges. In a nutshell, it is a development environment for creating automated, highly parallelized solutions that are natively capable of powerful healthcare EDI translations. This implementation is customized for quickly-provisioned Oracle Cloud Instances, and utilizes an Oracle database (or ADW/ATP instance) for operation, but can send and retrieve data to and from Microsoft SQL Server and Oracle databases.

The underlying automation system is driven by a customized PowerShell Core 7 environment with over 60 additional "cmdlets" that control the automation environment as well as the SERENEDI translation engine. The translation engine is capable of transforming EDI to and from CSV, XML, and two different types of database storage systems. The predefined field names are the mappings that bind data elements to the HIPAA EDI transaction sets. These are defined within the CGIF2 mapping technology, detailed here in this manual.

To help offset the learning curve needed to create *outbound* EDI transactions, the SERENEDI distribution database ships with the sample data and stored procedures that are used to generate the 14 "seed" files, one for each of the supported 5010 transaction sets. To query data on *inbound* EDI transactions, the built-in pipeline system has a rich set of capabilities for transforming hundreds or thousands of EDI files into queryable database tables. This pipeline is the default automation process for transforming EDI files into two-dimensional BIN tables, and is an easy place to start learning how the SERENEDI engine works and what it does – you can start merely by dragging EDI files into the 01_in_edi folder and it will start working from there. As the automation pipeline encounters new fields from the incoming data, the BIN tables are automatically extended with new columns to hold this data.

The advantage of the Flat BIN system is that the data is very accessible; the disadvantage is that the denormalized nature of the data makes it a bit more difficult to access some of the data that is stored that way. The Hierarchical Database is another BIN system that trades accessibility for flexibility. Loops in the EDI transaction are stored in separate database tables, but the cost is in complexity: you will need to know how each table relates to the parent table in order to parse it. Data is stored much more efficiently with this system and takes less time to transfer.

Along with these database capabilities, SERENEDI contains a rules engine to measure EDI compliance with the rules defined within the HIPAA Implementation Guides. This rules engine is user-extensible – if you need to add custom rules to analyze incoming EDI transactions, the simple REP Code language is documented here to allow you to create new rules.

We hope this user manual clearly presents the capabilities of the SERENEDI engine, and we aspire to ensure that by using these tools, you will be able to adapt to new challenges in the healthcare industry quickly and effectively.

Technical Summary

FEATURE	CAPABILITY
Operating Systems	The SERENEDI OCI Version is pre-installed to an Ubuntu 20 instance.
Database Compatibility	SERENEDI OCI Version requires an Oracle database for operation; EDI data can be stored and retrieved from SQL Server and Oracle databases.
Transformations	SERENEDI can translate EDI files to and from CSV files, XML files, and both flat and hierarchical database formats.
GUI	SERENEDI includes an a Blazor-based, web browser-based interface that enables testing of transforms, maintaining triggers and events, BIN activities, database endpoints, and SecureFTP sessions.
Specifications Supported	5010 270/271, 276/277, 277 CA, 278 REQ & 278 RESP, 820, 820X, 824, 834, 835, and 837 I & 837 P. Integrity checking of SNIP Types 1 and 2 is supported for all of these transactions, whereas 834/835/837 I/837 P support deeper SNIP Type 5 integrity checks with a user-extensible rules engine.
Automation	Events are generated based on file-system triggers or SQL-based triggers, or based on date and time criteria.
Scripting System	SERENEDI events are handled by a custom PowerShell Core environment with custom extensions. Visual Code IDE debugging is supported.

Operating System

SERENEDI is pre-installed on Ubuntu 20.02. It uses the default Ubuntu 20 Oracle Cloud Image with the following additions:

XFCE – This is a low-resource desktop environment needed for launching FireFox (for the Blazor GUI) and Visual Studio Code

Tiger VNC – This allows the graphics environment to be launched through the SSH shell.

Chromium – This is a web browser that is pre-setup to launch the locally hosted SERENEDI Studio GUI.

Visual Studio Code – This is the GUI used for creating and debugging SCORE Scripts.

All of the above can only be accessed with access to the private SSH Key, and all VNC traffic must be tunneled through this SSH session as described in the Installation instructions. This guarantees a secure environment for processing your PHI.

Database Compatibility

SERENEDI requires an Oracle Database Server instance to run the serenediCore distribution database. This contains all the core tables SERENEDI needs to function. Because SERENEDI does not utilize columns greater than 30 columns, it *should* be compatible with legacy Oracle databases that are able to be accessed by the ODP.NET Managed Database Access library maintained by Oracle.

By default, SERENEDI sends decoded EDI information submitted to its Decode pipelines to the serenediCore distribution database. It is possible to send this data to an external SQL Server or Oracle database by configuring database endpoints within the environment and directing pipelines to use this endpoint.

Transformations

Out of the box, SERENEDI can transform the supported HIPAA EDI transactions to flat database tables, hierarchical database tables, XML files, and CSV files. Because SERENEDI has every valid HIPAA EDI element built in and pre-mapped, this capability is built-in and nonconfigurable. The output of one EDI conversion is a complete representation of all data elements within the original file, and the bidirectional capability of SERENEDI means you should be able to reverse the transformation to create a binary-accurate representation of the original file.

EXAMPLE:

As a quick introduction, let's say you need to get all the unique subscriber last names from two gigabytes of mixed 837 Professional files. Starting from a fresh SERENEDI installation, you would need to do the following:

1. Drag all the files into the <install folder>/serenedi/pipeline/006_EDIToBIN/01_in_edi.
2. Wait for them to finish processing.
3. Run the following query:

```
SELECT DISTINCT(L2010BA_NM103PERSN_LNM) FROM BIN_5010_837P
```

The 006 Pipeline demonstrated in this example is ideal for data warehousing and querying the data as simply as possible – it will stuff all the EDI's transaction data into a two-dimensional data table, and will also automatically expand that table's schema when new EDI fields are encountered. There are two main tradeoffs with using this method: first, these two-dimensional projections of a hierarchical data source store data very inefficiently; and second, extracting certain data elements can be difficult because header-level elements are repeated across database rows.

The 007 Pipeline stores hierarchical data by assigning one database table for every loop defined by the implementation guides, with rows of each table identified by a unique ID specific to the file as well as a parent-child key mechanism that preserves the data aggregation present within the original EDI file. Thus, all line items stored in the 2400 loop table are keyed to one row in the 2300 claim table, and so on up to the ISA loop header table. This eliminates data redundancy, but also makes it more difficult to make ad-hoc queries of the data, as you need to know how these tables relate to one another.

On the other side of the coin is *encoding,* creating new EDI files from your enterprise data. The learning curve here is steeper, as you need to become very familiar with the column-naming convention SERENEDI uses to project this data. The distribution database ships with a number of SQL-stored procedures that pull some sample, fake PHI together in a way that can be fed to SERENEDI and generate the sample EDI files. These "seed" files can be used as a starting point for developers creating their own EDI extracts. One key point to remember here is that *decoding* can be done without development. If you are challenged to find a way to make SERENEDI encode a certain sequence of segments, it is easy to manually create an EDI file with these segments and then decode it to see how SERENEDI translates the information. Because this is a bidirectional transformation, you learn the exact data and mappings you would need to supply to get SERENEDI to encode it.

Graphical User Interface

SERENEDI ships with a graphical user interface (GUI) called SERENEDI Studio to enable you to test various transformations and see the results in real time. This web app allows you to test database extracts you have created using database development tools (such as SQL Server Management Studio or Oracle SQL Developer applications). This way, you gain immediate feedback about the validity of your mappings and logic.

In addition to this, SERENEDI Studio enables you to create and maintain *triggers* that control the highly parallel automation system, SecureFTP *sessions* that enable SFTP access, and database *endpoints* that enable access to databases outside of the distribution database. Furthermore, it allows you to see the status of BIN items that have been committed to the BIN system.

When you use SERENEDI Studio to maintain SERENEDI objects, it does not directly make writes to the distribution database – instead, it leverages the SCORE scripting commands to make these changes, and it provides a record of these commands in the *runbox* on the main screen of the interface. This fulfills a second role of SERENEDI Studio: to help end-users learn the commands that drive the automation environment.

Specifications Supported

SERENEDI deeply targets the transactions used most in the healthcare industry: 5010 270/271, 276/277, 277 CA, 278 REQ & 278 RESP, 820, 820X, 824, 834, 835, and 837 I & 837 P.

When decoding EDI files, the SERENEDI Engine will check basic syntax covering segment repeats, loop repeats, and segment composition. For the 834 Eligibility, 835 Remittance, and 837 Institutional and Professional specifications, SERENEDI has a user-extensible rules engine covering over 300 distinct validation rules. The engine will optionally check code-set compliance during decoding, covering claim adjustment reason codes, remittance adjustment reason codes, ICD 9 & 10 CM & PCS code sets, claim frequency codes, provider taxonomy, state abbreviations, and ISO country codes. Updates for these code sets are downloaded from the SecureFTP site that distributes the SERENEDI binaries.

Automation

SERENEDI uses a *multi-process* model for automation. Triggers are scanned by a background service, and when firing criteria are discovered, new events are created. A number of background worker processes poll the events table, and when they discover new work to do, they take ownership of that event and run a SCORE script – a concatenation of SERENEDI and PowerShell Core. When the SCORE script has finished executing, any messages regarding errors that occurred along the way are inserted into the messages table. If a critical error occurs at any point, the event is flagged as such at the high-level events table so it can be more easily caught and researched.

Scripting System

SERENEDI uses PowerShell Core for its versatility and cross-platform compatibility. PowerShell Core scripts (“SCORE” scripts) spawn an instance of the SERENEDI translation engine and contain all the dozens of custom commands that work with that engine. Normally these scripts are executed sight-unseen using parameters passed to it by the automation system. However, it is also possible to run the script within a Visual Code interactive shell and to run it line by line, provided a few commands are inserted at the top of the SCORE script that initializes the SERENEDI engine. In this way, SERENEDI leverages proven, open-source development and debugging environments to speed the development of complex business processes.

If a particular integration task is not provided within the SCORE scripting environment, it’s quite possible to extend the functionality by linking to outside libraries for many tasks.

OCI Installation

INSTALLATION

SERENEDI is installed in three phases: **Prerequisites, Main Install,** and **Activation**.

The **Prerequisites** phase sets up the Ubuntu 20 instance to have the secure GUI necessary for the Visual Studio Code environment as well as the serenediStudio Web App. No external firewall ports are opened for this process; all access still requires the Oracle Instance SSH private key using a VNC port that is opened with a "bridge" on the client side. This step is optional, as SERENEDI can still operate in a pure server mode.

The **Main Install** phase installs all of the components necessary for SERENEDI, and creates the SERENEDI distributions folders. To complete this step, the end-user must set up a valid Oracle connection string (and possibly the wallet for ADW/ATP Autonomous databases) and successfully test it from the command-line.

The **Activation** step deploys the SERENEDI distribution database to the Oracle server and activates the SERENEDI service. This completes the installation.

PREREQUISITES

a. Create an instance using the pre-defined Oracle-supplied Canonical Ubuntu 20 image. Do *not* choose the minimal installation. When configuring the Boot Volume, make sure to select the "Use In-Transit Encryption" so that all traffic to and from the CPU to the boot volume remains encrypted end-to-end. Make sure download the SSH keys. You can use most shapes that provide at least 8GB of RAM or above. Note that if you are installing an evaluation instance of SERENEDI, it will limit utilization to a maximum of four cores.
b. Log in to the instance using a terminal that allows binary file uploads/downloads. The SSH terminal SmarTTY is one good option that is freely available.
c. Upload the "install.zip" file to the default ubuntu account.
d. Enter the following:

```
sudo apt install unzip
unzip install.zip
cd install
chmod +x *.sh
sudo ./setup01.sh
```

e. The unattended setup script will take approximately half an hour to run on a single core instance. It will automatically reboot the instance when the script completes.
f. After the system reboots and you reconnect your terminal session, you will need to set up the VNC password. Enter the following:

```
vncpasswd
```

g. Enter a password twice to set the VNC password. The password is limited to 8 characters.
h. Enter the steps to enable and start the VNC server:

```
sudo systemctl enable vncserver@1
sudo systemctl start vncserver@1
```

i. Next, you will need to enable access to the local 5901 port on your instance via an SSH bridge. This ensures that all VNC/GUI traffic is using secure encryption. Type the following on your *local workstation*, replacing the ssh.key with the path and filename of the private SSH key you downloaded when creating your Ubuntu 20 instance, and at the end place the IP of the OCI Instance:

```
ssh -i <<ssh.key>> -L 61000:localhost:5901 -C -N -l ubuntu <<IP of OCI Instance>>
```

j. You will be prompted to say "yes" or "no" when shown the SSH signature of the OCI instance; select Yes.
k. Next, you will need to launch a VNC client (such as the freely available TightVNC client) and open the following location:

```
localhost:61000
```

l. You should be prompted for the password you entered in the step above. If everything is successful, you will see the green Ubuntu Mate desktop, and a single Visual Studio Code icon.
m. Launch Visual Studio Code, and wait for it to launch. Then, press Ctrl-Shift-X to go to the Extensions Search pane, and type in "powershell" and enter.
n. The top result should be "PowerShell" from Microsoft. Click on it, and then in the right pane, click "Install".

This completes installing the Prerequisites necessary to use the serenediStudio GUI and Visual Studio Code functionality of SERENEDI.

MAIN INSTALL

a. Run the following:

```
sudo ./setup02.sh
```

b. Create the schema for the SERENEDI Distribution database by logging into your Oracle database with administrator privileges and execute the following. Note that the schema name *serenedi* is not hard-coded; you can substitute any schema name you wish. Enter the following:

```
CREATE USER serenedi IDENTIFIED BY <<SERENEDI PASSWORD>>;
GRANT CREATE SESSION TO serenedi;
GRANT DWROLE TO serenedi;
GRANT CREATE TABLE TO serenedi;
GRANT UNLIMITED TABLESPACE TO serenedi;
```

GRANT DWROLE is especially for ATP/ADP Autonomous cloud databases. For other databases, you may replace it with:

```
GRANT ALL PRIVILEGES TO serenedi;
```

Next, create a new connection that uses the above user and password, and execute the database script.

Set up the Connection String

For ADP/ATP Oracle Autonomous Databases:

Download the wallet.zip file for the connection and unzip the contents to /opt/serenedi/bin/wallet.

Then, edit the existing file within /opt/serenedi/bin/cnnstr.txt:

```
"ORACLE","User id=serenedi;Password=<<Your Password Here>>;Pooling=False;Data
Source=<<Data Source Here>>;"
```

The data source will depend on your ADW/ATP instance, and what level of utilization priority you wish to assign.

For Oracle Databases:

For EZ Connect database connections, edit the /opt/serenedi/bin/cnnstr.txt as follows:

```
"ORACLE","Data Source=<<ORACLE DOMAIN ADDRESS>>:1521/xepdb1;DBA Privilege=SYSDBA;User Id=serenedi;Password=<<SERENEDI PASSWORD>>;"
```

In this example, we are using the EZ Connect specification to specify the Oracle Domain, the default port (1521), and the service being used (xepdb1), followed by the user name and password information.

For other examples of Oracle connection strings, see:

https://www.connectionstrings.com/oracle-data-provider-for-net-odp-net/

Note that the connection string needs to be in the second set of quotes, and the first set of quotes is fixed with "ORACLE".

To test the connection, run the following:

```
cd /opt/serened/bin
dotnet SERENEDI2.dll TESTDB
```

If the return result is "DATABASE: OK" then the database connection is valid, and this phase of installation is complete.

ACTIVATION

a. To create the SERENEDI Schema and complete the installation process, run the following:

```
sudo ./setup03.sh
```

To immediate test that the installation is successful, enter:

```
dotnet SERENEDI2.dll LICENSE
```

Launch the GUI Environment

SERENEDI Instances run a lightweight GUI environment via VNC that is tunneled through the SSH connection. This means you must have the SSH Private Key for the OCI instance in order to enter the GUI. You must have a VNC client on your local desktop in order to use the GUI on the SERENEDI instance.

First, you must set up an SSH bridge. On Windows or UNIX PCs, run the following:

```
ssh -i <<ssh.key>> -L 61000:localhost:5901 -C -N -l ubuntu <<Public IP of OCI Instance>>
```

The first bolded option is the filename to the private SSH key, as downloaded from Oracle when obtaining the SERENEDI OCI instance. The 61000 here is not fixed; it should be any local open IP port on the your PC. The last bold option is the public IP address of the SERENEDI OCI instance.

Then, launch a VNC instance targeted at localhost:61000. The password is: **ceti2020pw**

The GUI is launched as the user "serenedi". The account is sudo-enabled with the password of "serenedi".

To launch SERENEDI Studio, click the green SERENEDI Icon.

OPTIONAL: Using Visual Code

To create new SERENEDI PowerShell Core Scripts (SCORE Scripts) using Visual Code, place these lines at the top of the script. This will register SERENEDI's commands with the environment and create a new session:

```
Set-Location /opt/serenedi/bin
Import-Module -Name (Resolve-Path 'SERENEDI2.dll')
sapi-InitializeSession -BaseDir '/opt/serenedi'
```

Make sure to remove/comment out the above lines of code when linking these scripts to Triggers in the automation environment; these are used *only* for Visual Code sessions.

There's also a way to use Visual Code to help troubleshoot your scripts using events fired during production. For example, if a specific trigger generated an event that critically errored and you're not able to immediately understand why, you have the option to load the environment registers for that completed event and step through your code line-by-line to understand how the error occurred.

To initialize the session for a specific event, change the third line to this:

```
sapi-InitializeSession -BaseDir '/opt/serenedi' -BizEventId <<Event ID here>>
```

SERENEDI Overview

This chapter will explain how to use the major features of SERENEDI. First, we'll review the business requirements behind the architecture. Then we'll explore the automation system, the SERENEDI engine that runs all workflows, and the BIN system that stores EDI data in a human-accessible form. We'll examine changes from Chiapas EDI Enterprise, and, finally, we'll walk through the Pipeline system. Those of you wishing to immediately get hands-on with the system can skip directly to the Pipeline section, as that also functions as a Quick Start guide to using SERENEDI.

The SERENEDI Technical Reference section goes much deeper into the details of the SCORE script commands, rules engine, and table schemas used in the distribution database.

Business Requirements

Before we go into the specifics of SERENEDI, we need to review the exact business objectives this technology is addressing. These are explained in the three guiding principles of SERENEDI's design, the Three P's:

PARALLELISM

Bottlenecks are very common in the healthcare IT space. When building new systems, legacy components can sometimes severely limit the potential to achieve objectives. For example, when building a claim processing system, you may find that a legacy eligibility-checking component is limited to one eligibility verification per second. This severely limits the systems you can build that rely on this mechanism.

SERENEDI is built from the ground up to handle multiple operations simultaneously, with special considerations to prevent database "locking" scenarios when storing data in the database. The error-logging and data-storage systems contribute to an automation environment designed to take full advantage of modern hardware.

PORTABILITY

The rise of Unix-based operating systems means that sometimes the computing resources available are far from homogenous – it's very common for IT server rooms to be filled with both Unix-based and Windows-based servers.

Furthermore, even without that consideration, most healthcare solutions are designed on a developer server, tested on a QA server, and released on a production server, meaning that the solution has to be migrated at least twice before running in production. If a solution is *portable*, it's easy to make this migration – and the less portable it is, the harder the process becomes.

SERENEDI makes it possible to develop a solution on a Windows-based workstation, test it on a Unix-based QA server, then release it on a Windows production server. A good example of this is the SERENEDI Pipeline system, which ships as a single SCORE script that quickly inflates to an entire directory heirarchy and over a dozen event triggers when the SERENEDI system is first started. This single script works the same on Windows and Unix, and because the script is broken up into different functional modules, it's easy to develop and maintain.

PROJECTION

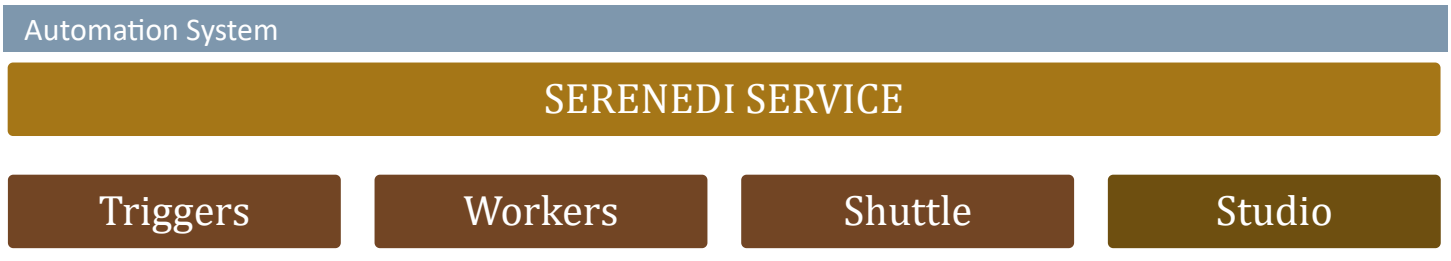
This software is primarily focused on one driving requirement: enable users to focus on the *business data* within HIPAA EDI transactions, and not on the transactions themselves. Because healthcare data requirements are so complex across the United States, the EDI formats themselves are also complex, so this is not easy to do.

There are a number of different technological solutions to this problem. Some parse EDI transactions with an XML schema and then let users map individual fields, one by one, to their own data requirements. Others give a fixed, preset translation capability and provide a number of database tables for the supported transactions; customization is often difficult for these solutions.

SERENEDI's approach is a bit different. Its proprietary technology *projects* EDI transactions directly into three formats: XML, database flat table, or database heirarchical table. This projection is bidirectional, and as long as SERENEDI's mapping rules are maintained and the file is completely HIPAA compliant, it can reproduce the original EDI transaction character for character. These projections are completely automated, "black box" operations that are built into SERENEDI.

If you are a developer who needs to extract business information from large amounts of existing EDI files, this capability lets you complete your work using only the existing pipelines as-is and accessing the data elements using SQL-stored procedures and views.

If you need to create new outgoing EDI files, you'll need to learn more about the CGIF2 (Chiapas Gate Intermediate Format Version 2) mapping system. It helps that you can always add new segments, decode them, and see how they are mapped in real time with the user interface tools provided. Furthermore, a collection of views and sample data enclosed with the distribution database keeps you from ever having to start completely from scratch.



Once SERENEDI is installed and linked to an instance of the distribution database, SerenediService starts up. This service is kept purposefully lightweight so it can run for many months without disruption and is responsible for launching other processes as needed. When it first starts, it assesses the cores allowed under the license and the physical cores available; whichever is *less* is the maximum number of worker processes allowed. This sets a ceiling on the simultaneous number of worker processes according to both the licensing and the limits of the server.

SerenediService handles the following tasks:

Trigger Scan

This process loads in the active triggers and polls the trigger criteria at a preset interval. When trigger-firing criteria are met, *events* are generated.

Worker

SERENEDI maintains a small number of workers that poll the distribution database constantly for new work in the form of events generated by trigger scans. When a large number of events are pending, SERENEDI scales up the number of simultaneous workers, but otherwise keeps it to a quarter of the maximum to minimize unnecessary database polling. Each event is tied to a SCORE script, which the worker runs with the arguments associated with that event. If a worker process exceeds four hours to execute a script, it is forcibly terminated and the worker slot is made available again.

Data Shuttle

The *data shuttle* is an ongoing process that enables the BIN system – a way to decode large amounts of EDI files into human-accessible database tables. Table schemas are dynamic in the BIN system and can expand according to the needs of incoming data. However, by ensuring that only the data shuttle process is moving data to the destination tables and altering schemas, it avoids the schema locks that would occur if multiple worker processes were trying to insert data and expand the schema at the same time.

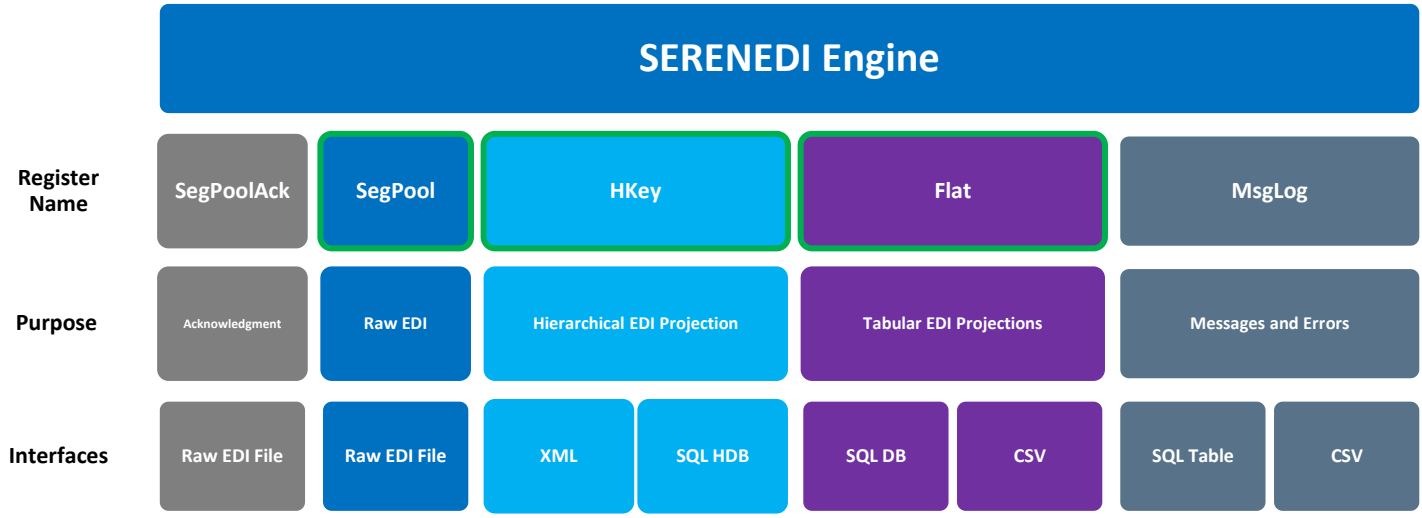
SERENEDI Studio

The SERENEDI Studio process is a web app that is accessible on the server machine and allows developers to test various data transformations as well as see the current state of the BIN, triggers, endpoints, and SFTP systems. This defaults to port 42000, and by default does not accept connections except from browsers launched on the server itself. The Studio process can be completely disabled and can be configured to accept outside connections.

SERENEDI Engine

The SERENEDI engine is the environment that is loaded every time an event is fired. Aside from the registers used for logging messages and generating Transaction Acknowledgments (defined within specification 999A1), there are three primary registers: SegPool, HKey, and Flat. These are shown in the following diagram. The green border indicates registers that support bidirectional projection, meaning that data can be transformed laterally between the three registers in addition to the methods listed on the Interfaces line.

The SERENEDI engine is composed of several parts: the registers, the interfaces, and the Integrity Rules Engine.



PRIMARY REGISTERS

SegPool

This register is the built-in representation of an EDI file. It interfaces directly with the file system to load and store EDI files.

HKey

This register represents an internal representation of an EDI file, laid out in hierarchical form. The process of projecting the SegPool to the HKey register is normally called *decoding*, whereas projecting from HKey to SegPool is *encoding*. HKey registers can be stored as XML files on the file system, to an internal XML register, or translated to a series of database tables where each table represents data stored in one loop.

Flat

This register represents a two-dimensional view of hierarchical data, with the data heavily repeated row after row. It is often the projected form of EDI data that is easiest to work with for normal operations.

AUXILIARY REGISTERS

SegPoolAck

This register stores a secondary SegPool completely dedicated to generating or parsing 999 Acknowledgment files. It provides a simple facility for generating transaction acknowledgments without needing to unload the primary SegPool register first.

MsgLog

This register stores all errors and messages accumulated during a workflow execution. The automation system will then normally push those messages to the BIZ_MSG table on the distribution database.

INTERFACES

Interface	Business Role
Raw EDI Files	These are the basic units of transaction containing PHI (Protected Health Information) between two enterprises.

XML	An XML-readable version of an EDI transaction enables NoSQL/XML-based database systems to ingest and process healthcare information. For enterprises that are accomplished in creating XML files, this is also an avenue to create transactions.
Hierarchical Database Tables	The HDB interface lets a transaction be split up into a number of tables, one for each loop within the transaction. Each row in each table is assigned a unique BIN ID for the whole transaction. Because the data is stored in a highly optimized way, it's ideal for high-speed decoding at the cost of a higher complexity of tracking the relationships between parent and child loops, which mirrors the way they are arranged in the implementation guides. For some transactions, creating files with logically arranged hierarchical tables may be easier than creating a flat projection of the data.
Flat Database Table	This is normally the easiest way to create or retrieve EDI transaction data, as all healthcare data for a transaction is projected into a single database table that can be queried or created. All the sample EDI transactions are created by a single stored procedure associated with each transaction to serve as a springboard for creating new transactions. This simplicity comes at the cost of highly denormalized data storage.
CSV File	This stores data in exactly the same way as the flat database table, except that it's stored in CSV files on the file system. It's provided for use with legacy enterprise systems that can only accept or export CSV data. Also, CSV files can easily be opened with Microsoft Excel.

INTEGRITY VALIDATIONS

Code Set Checks

SERENEDI does not check for code-set compliance by default. The SCORE Script command **sapi-SegPoolToHKey**, which decodes incoming EDI files can be made to do so with a command-line flag **-EnableCodeSetChecks**. The following code sets are validated during decoding:

Code Set	Source
Claim Adjustment Reason Codes	Washington Publishing Company
Remittance Adjust Reason Codes	Washington Publishing Company
Claim Frequency Codes	Washington Publishing Company
ICD-9-CM Diagnosis Codes	CMS
ICD-9-PCS Procedure Codes	CMS
ICD-10-CM Diagnosis Codes	CMS
ICD-10-PCS Procedure Codes	CMS
National Drug Codes	CMS
Provider Taxonomy Codes	Washington Publishing Company
State Abbreviation Codes	United States Post Office
ISO Country Codes	United States Post Office

These code sets are updated by their sources periodically, and SQL Server update scripts that set these to their latest versions in the distribution database are available in the Chiapas EDI SecureFTP download site under the Code Set folder.

Integrity Rules Engine

SERENEDI is capable of five levels of SNIP integrity checks for incoming EDI file decoding operations. The SNIP Type checks are summarized here:

1. Integrity Testing – Basic segment and element integrity checks
2. Requirement Testing – Validating the presence of elements, segments, and loops marked as mandatory

3. Balance Testing – Testing of monetary totals across loops and transactions
4. Situational Testing – Testing of specific inter-segment situations
5. External Code Set Tests – Testing specific values being present in predefined code sets
6. Line of Business Testing

SERENEDI runs SNIP Type 1 and 2 validations automatically across all file decodes – this cannot be disabled. SNIP Type 5, External Code Set Tests, is an optional integrity check that executes during decoding. SNIP Type 3 and 4 rules are parsed using the built-in Integrity Rules Engine. These rules can only be executed after the file has been successfully decoded.

If you need to add SNIP Type 6 Line of Business rules for a particular trading partner, you have the ability to add new rules with the user-extensible rules engine.

BIN System

SERENEDI's data storage system was designed according to the following business requirements:

1. Provide a scalable solution to ingest large amounts of EDI data across multiple SQL Server or Oracle databases
2. Make this EDI data easy to access with common SQL queries and stored procedures
3. Make the data accessible in two ways:
    a. The *Flat* BIN system is a two-dimensional, denormalized representation of heirarchical data that is the easiest to use for reading EDI data or for encoding simpler EDI files.
    b. The *Hierarchical* BIN (HDB) system stores data using tables in which data from every loop is stored in separate, linked tables. It stores data more efficiently and makes it easier to encode complex situations, at the cost of needing to track the interrelationships of the tables.

By default, SERENEDI ships with two pipelines that automatically decode files to the distribution database, 006_EDIToBIN and 007_EDIToHDB. Unless overridden, they will go to data tables that are named according to the specification of the data. An incoming 837P file will go to BIN_5010_837P for that Flat interface. For the HDB interface, it will decode to a number of tables matching the loops it contains: HDB_5010_837P_ISA, HDB_5010_837P_GSHDR, and so on.

When a workflow pulls in data during execution, the data is not immediately inserted into these destination tables. Instead, it goes to a temporary table named according to this convention:

Flat: T_<event Id>_<10-digit random number>

HDB: T_<event Id>_<increment>_<10-digit random number> for each loop in the incoming file

The background *data shuttle* process transports the data from the temp tables to the actual BIN table destinations. Unless overridden, by default, the data shuttle will alter the destination table schemas to add new data elements when they are encountered. That is, if the existing database schema does not hold the data elements needed to accommodate the information coming with the new incoming EDI file, it will be expanded so the new data can be stored. This is called a *ForceMerge* operation. The *Merge* operation will prevent the data shuttle from expanding the schema, and instead raise messages when data is encountered that cannot be stored in the existing schemas.

The data shuttle ensures that data can be streamed into a database from literally over two dozen active operations, and that the data will all reach its destination in the same way. This fulfills requirement 1.

Requirement 2 is fulfilled by having all of the decoded data in a form that is fairly straightforward to access using normal SQL. Requirement 3 is fulfilled by the two methods of data storage allowed by the BIN system.

NOTE: The background data shuttle process handles all schema alterations in real time, without needing user intervention. However, this process can be "frozen" by external factors if you use database cursors to open the BIN tables. The cursors could open up a *schema lock* that prevents the schema from being changed while the cursor is reading data. This can severely delay the speed at which SERENEDI processes data. The best practice is to access the BIN and HDB data using only set-based operations. If you need to iterate the data in a cursor, copy the rows you need into a temporary table, and then iterate those records – this way, the main BIN tables will not be locked by a long-running read operation.

Pipelines

The built-in pipeline system is designed to be a highly accessible, easy-to-use method to access the core conversions without needing to directly program the SERENEDI system. Here, a *pipeline* specifically means a business process associated with both a SCORE script and one or more triggers in the automation system. Generally, SCORE scripts have code to "bootstrap" the environment – that is, to set up the folders and triggers.

After SERENEDI is set up but prior to the background SERENEDIService being activated, there is only a single SCORE Script named Pipeline.ps1 in the Pipeline folder, as the bootstrap sequence has not been executed yet. Once the service is started, the SCORE script will run the bootstrap sub-module and create a number of both folders and triggers. All execution flow goes into that same Pipeline.ps1 SCORE script, but different parameters are used to execute the various triggers.

In most cases, actually using the pipelines is pretty simple – just drop new files into the incoming folder for each pipeline. The action of moving the file from the *in* folder to the *out* folder "unlocks" the file and gets it ready to be processed. If there is a critical integrity error in the file, the source file is moved from the 02_done folder to the 04_err folder.

QUICK START

To quickly demo the functionality of SERENEDI, you can execute the following steps. First, from a file explorer window on the SERENEDI server, make a copy of the SERENEDI/seed folder and put the copies into the first incoming folder, C:\serenedi\pipeline\001_Normalize\01_in_edi (base folder depends on the installation option). Then, move the 14 files that result from this operation, that are generated in the 03_out_edi folder, into the *in* folder of the next pipeline in sequence. Skip Pipeline 006 and drop the results of the *out* folder from Pipeline 005 directly to the *in* folder for Pipeline 007. Wait one minute for the data to be completely finalized in the BIN system.

Then, open the user interface from the local server via http://127.0.0.1:42000 (default install option), go to the Endpoints screen, select the left menu bar, and then enter the following in the AD-HOC SQL window:

```
INSERT INTO BIZ_EVENT (BIZ_TRIGGER_ID, EVENT_DATA1, EVENT_DATA3) SELECT (SELECT BIZ_TRIGGER_ID FROM
BIZ_TRIGGER WHERE TRIGGER_NAME = 'PIPE008_BINToEDI'), BIN_ID, 'PIPE008_BINToEDI' FROM BIN_LOG WHERE
BIN_STATUS = 'COMPLETE' AND BIN_TYPE = 102
```

The above actions will convert all seed files to CSV files, and then back to EDI files; convert the results to XML files, and then back to EDI files; convert the results of that operation to the HDB BIN system; and finally, pull the HDB data back into EDI files, completing a tour of all of the supported *projection* operations of the engine. As long as the original files are HIPAA compliant, they should still be the same size and content as the files resulting from the first Normalize operation.

Pipeline 001: Normalize

Pipeline	Type	Purpose
001: Normalize	Upload	Reformat incoming EDI files so that they have consistent separator characters and line feeds between segments.

Initial: \serenedi\pipeline\001_Normalize\01_in_edi
Finished:\serenedi\pipeline\001_Normalize\02_done_edi

```
Output:   \serenedi\pipeline\001_Normalize\03_out_edi
Error:    \serenedi\pipeline\001_Normalize\04_err_edi
```

Parameters:

```
EVENT_DATA1: full path to filename of the EDI file
EVENT_DATA3: PIPE001_NORMALIZE
```

This pipeline decodes and then immediately *re-encodes* an EDI file. It will set all segment and element terminators to default values, and add a carriage return and line feed after every segment. Any syntactically invalid elements that do not pose a critical error will be discarded – for example, a qualifier/identifier pair where either the qualifier or the identifier is missing. This "normalizing" process is useful to ensure a set of files have homogenous formatting and can be easily read in a text editor.

Pipeline 002: CSVFromEDI

Pipeline	Type	Purpose
002: CSVFromEDI	Upload	Convert incoming EDI files to CSV format.

```
Initial:  \serenedi\pipeline\002_CSVFromEDI\01_in_edi
Finished: \serenedi\pipeline\002_CSVFromEDI\02_done_edi
Output:   \serenedi\pipeline\002_CSVFromEDI\03_out_csv
Error:    \serenedi\pipeline\002_CSVFromEDI\04_err_edi
```

Parameters:

```
EVENT_DATA1: full path to filename of the EDI file
EVENT_DATA3: PIPE002_EDIToCSV
```

This pipeline converts incoming EDI files to a CGIF2-formatted CSV file. The resulting file will enclose all values within quotes and have a header row containing the column names. The first column name will be prefixed by the specification tag. The destination filename will be the same as the source filename, except with a .CSV extension.

Pipeline 003: CSVToEDI

Pipeline	Type	Purpose
003: CSVToEDI	Upload	Convert incoming CGIF2 flat-formatted CSV files to EDI.

```
Initial:  \serenedi\pipeline\003_CSVToEDI\01_in_csv
Finished: \serenedi\pipeline\003_CSVToEDI\02_done_csv
Output:   \serenedi\pipeline\003_CSVToEDI\03_out_edi
Error:    \serenedi\pipeline\003_CSVToEDI\04_err_csv
```

Parameters:

```
EVENT_DATA1: full path to filename of the CSV file
EVENT_DATA3: PIPE003_CSVToEDI
```

This pipeline executes the reverse of the 002 pipeline, converting a CGIF2-formatted CSV file back to EDI. If the file cannot be successfully converted to EDI, it will be placed in the 04_err folder. The EDI file will inherit the same filename as the CSV file, except with a .txt extension.

Pipeline 004: XMLFromEDI

Pipeline	Type	Purpose
004: XMLFromEDI	Upload	Convert incoming EDI files to CGIF2-formatted XML files.

```
Initial: \serenedi\pipeline\004_XMLFromEDI\01_in_edi
Finished:\serenedi\pipeline\004_XMLFromEDI\02_done_edi
Output:  \serenedi\pipeline\004_XMLFromEDI\03_out_xml
Error:   \serenedi\pipeline\004_XMLFromEDI\04_err_edi
```

Parameters:

```
EVENT_DATA1: full path to filename of the EDI file
EVENT_DATA3: PIPE004_XMLFromEDI
```

This pipeline projects incoming EDI files into CGIF2-formatted XML files. The files generated will begin like this:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<CGIFXML3Root_U0 xml:space="preserve">
… edi data...
</CGIFXML3Root_U0>
```

The **U0** will be replaced with the actual specification tag for this file.

Pipeline 005: XMLToEDI

Pipeline	Type	Purpose
005: XMLToEDI	Upload	Convert incoming XML files to EDI.

```
Initial: \serenedi\pipeline\005_XMLToEDI\01_in_xml
Finished:\serenedi\pipeline\005_XMLToEDI\02_done_xml
Output:  \serenedi\pipeline\005_XMLToEDI\03_out_edi
Error:   \serenedi\pipeline\005_XMLToEDI\04_err_xml
```

Parameters:

```
EVENT_DATA1: full path to filename of the XML file
EVENT_DATA3: PIPE005_XMLToEDI
```

This pipeline converts incoming XML files to EDI. If the file does not adhere to the CGIF XML rules, then the conversion will fail and the XML file will be placed in the 04_err folder. The filename will be the same as the source XML except with a .txt suffix.

Pipeline 006: EDIToBIN

Pipeline	Type	Purpose
006: EDIToBIN	Upload	Load an incoming EDI file to the Flat BIN system.

```
Initial: \serenedi\pipeline\006_EDIToBIN\01_in_edi
Finished:\serenedi\pipeline\006_EDIToBIN\02_done_edi
Error:   \serenedi\pipeline\006_EDIToBIN\03_err_edi
```

Parameters:

```
EVENT_DATA1: full path to filename of the EDI file
EVENT_DATA3: PIPE006_EDIToBIN
```

This pipeline ingests files into the Flat BIN system built into SERENEDI. If the BIN table does not exist, it will be created. If the table exists but certain mapped fields are not present, the fields will be added. The EDI file will be assigned a unique BIN_ID defined in the BIN_LOG table.

The default table name is BIN_5010_<specification short name> *(example: BIN_5010_837P)*

This pipeline is geared toward ingesting large amounts of data into the BIN system, so it doesn't wait for the data to go into the destination tables – instead, it places the data into a marked temporary table and immediately exits. To see if the data for this event is finalized in the BIN system, you'll need to look up the BIZ_EVENT_ID in the BIN_LOG table and see if the BIN_STATUS is COMPLETE. When that occurs, the data is available in the destination BIN table, which is listed in the BIN_TABLE field.

Pipeline 007: EDIToHDB

Pipeline	Type	Purpose
007: EDIToHDB	Upload	Loads an incoming EDI file to the HDB BIN system.

```
Initial: \serenedi\pipeline\007_EDIToHDB\01_in_edi
Output:  \serenedi\pipeline\007_EDIToHDB\02_done_edi
Error:   \serenedi\pipeline\007_EDIToHDB\03_err_edi
```

Parameters:

```
EVENT_DATA1: full path to filename of the EDI file
EVENT_DATA3: PIPE007_EDIToHDB
```

This pipeline ingests files into the HDB BIN system built into SERENEDI. The HDB stores data hierarchically, with one table present for each loop and all tables joined by keys.

The default table names are HDB_5010_<specification short name>_<loop short name>

For example, the seed 834 file will decode to the following tables:

HDB_5010_834_ISA
HDB_5010_834_GSHDR
HDB_5010_834_STHDR
HDB_5010_834_L1000A

HDB_5010_834_L1000B
HDB_5010_834_L2000
HDB_5010_834_L2100A
HDB_5010_834_L2300

These tables will be prefixed by two to four fields, depending on the situation. For ISA tables, only the BIN_ID and BIN_IX (BIN Index) fields are used. For all other tables, a PAR_BIN_IX relates that loop to its parent loop table. For L2300 tables for 837 Institutional and 837 Professional specifications only, an optional PAR_2000C_IX field also relates claims to a specific 2000C patient loop.

Similar to what was described in the previous pipeline for the Flat BIN system, this pipeline will place data into a number of temp tables before actually populating the destination tables. Once all the temp table data has been migrated to the destination tables by the background data shuttle process, the BIN_LOG entry for this file will be changed to COMPLETE.

Pipeline 008: BINToEDI

Pipeline	Type	Purpose
008: BINToEDI	Upload	Retrieves EDI data from either the Flat or HDB system identified by the BIN_ID and encodes it to an EDI file to the filesystem.

Output: \serenedi\pipeline\008_BinToEDI\01_out_edi

Parameters:

EVENT_DATA1: BIN_ID of the file to pull from the BIN system, **or** a SELECT or EXEC SQL statement for a dynamically generated FLAT-formatted extract
EVENT_DATA2: **OPTIONAL:** full path and filename of the EDI file to create
EVENT_DATA3: PIPE008_BINToEDI

This pipeline will pull data from a completed Flat BIN or HDB BIN and create a new EDI file. By default, the filename will follow the BIN_FILENAME listed in the BIN_LOG table and be placed in the 01_out_edi folder. This can be overridden by placing a fully pathed filename in EVENT_DATA2. Files that are encoded in either the Flat BIN or HDB BIN can be extracted with this pipeline.

Here's an example of the SQL you would need to execute to generate a single file that was previously decoded to the BIN system:

```
INSERT INTO BIZ_EVENT(BIZ_TRIGGER_ID, EVENT_DATA1,EVENT_DATA3) SELECT
BIZ_TRIGGER_ID,<<BIN ID>>,'PIPE008_BINToEDI' FROM BIZ_TRIGGER WHERE
TRIGGER_NAME='PIPE008_BINToEDI'
```

To supply a name and location for this file and override the defaults, it should be supplied in EVENT_DATA2 as follows:

```
INSERT INTO BIZ_EVENT(BIZ_TRIGGER_ID, EVENT_DATA1, EVENT_DATA2, EVENT_DATA3) SELECT
BIZ_TRIGGER_ID,<<BIN ID>>,'c:\SERENEDI\OUTPUT.TXT', 'PIPE008_BINToEDI' FROM
BIZ_TRIGGER WHERE TRIGGER_NAME='PIPE008_BINToEDI'
```

If a non-numeric value is supplied in EVENT_DATA1, this pipeline will assume it is dynamic SQL tied to the distribution database. Here's an example of regenerating the seed_837p.txt file from the sample data built into the system:

```
INSERT INTO BIZ_EVENT(BIZ_TRIGGER_ID, EVENT_DATA1, EVENT_DATA2, EVENT_DATA3) SELECT
BIZ_TRIGGER_ID,'EXEC USP_837P_EXTRACT','c:\SERENEDI\SEED_837P.TXT', 'PIPE008_BINToEDI'
FROM BIZ_TRIGGER WHERE TRIGGER_NAME='PIPE008_BINToEDI'
```

Pipeline 009: Integrity

Pipeline	Type	Purpose
009: Integrity	Upload	Decodes an EDI file with all integrity and codeset checks enabled.

```
Initial: \serenedi\pipeline\009_Integrity\01_in_edi
Finished:\serenedi\pipeline\009_Integrity\02_done_edi
Output:  \serenedi\pipeline\009_Integrity\03_integ_html
Error:   \serenedi\pipeline\009_Integrity\04_err_edi
```

Parameters:

```
EVENT_DATA1: filename of the EDI file to check
EVENT_DATA3: PIPE009_INTEG
```

This pipeline decodes an incoming EDI file with full code-set checks, and then executes the Integrity Rules Engine to perform a deeper rule validation on the file. This pipeline is currently supported only for specifications 834, 835, 837 P, and 837 I.

Pipeline 010: Event

Pipeline	Type	Purpose
010: Event	Upload	Creates an event based off of an XML file, then writes messages to an output XML file sharing the same filename with a .result suffix.

```
Initial: \serenedi\pipeline\010_Event\01_in_xml
Finished:\serenedi\pipeline\010_Event\02_done_xml
Output:  \serenedi\pipeline\010_Event\03_msg_xml
Error:   \serenedi\pipeline\010_Event\04_err_xml
```

Parameters:

```
EVENT_DATA1: filename of the XML file containing EVENT information
EVENT_DATA3: PIPE010_EVENT
```

This pipeline allows you to create a new event in the automation system using a specially crafted XML file. It will parse out the arguments of the XML file, insert them into the automation system, wait for completion, and then output the messages generated by that workflow in the 03_msg_xml folder. The message result file bears the original filename with a .result suffix added to the filename.

Pipeline: SFTP_MIRROR

Pipeline	Type	Purpose
SFTP_MIRROR	SQL	Provide file mirroring for all triggers linked to a SecureFTP session.

This is an SQL-triggered pipeline that periodically *mirrors* the local and remote file systems for triggers that are tied to a SecureFTP session. This mirroring operation can be tested with the SERENEDI Studio GUI under the SFTP Session menu bar.

It works by actively scanning the BIZ_TRIGGER table for passive, enabled triggers that are linked to a SecureFTP session. These SFTP mirror triggers need to have the local synchronization directory in the FORCE_ARG3 field, and the remote synchronization directory in the FORCE_ARG4 field. The LAST_POLL_DT and POLL_INTERVAL are used by this pipeline to control the polling of these SecureFTP directories. The actual direction of the synchronization is controlled by the SFTP session itself.

Since SFTP operations can take a minute or more, it's best to set the POLL_INTERVAL on the SFTP triggers to 900 or more (15+ minutes) so the remote SecureFTP server is not bombarded by refresh requests.

SERENEDI Studio

SERENEDI Studio is the graphical user interface for SERENEDI. It fulfills a number of requirements:

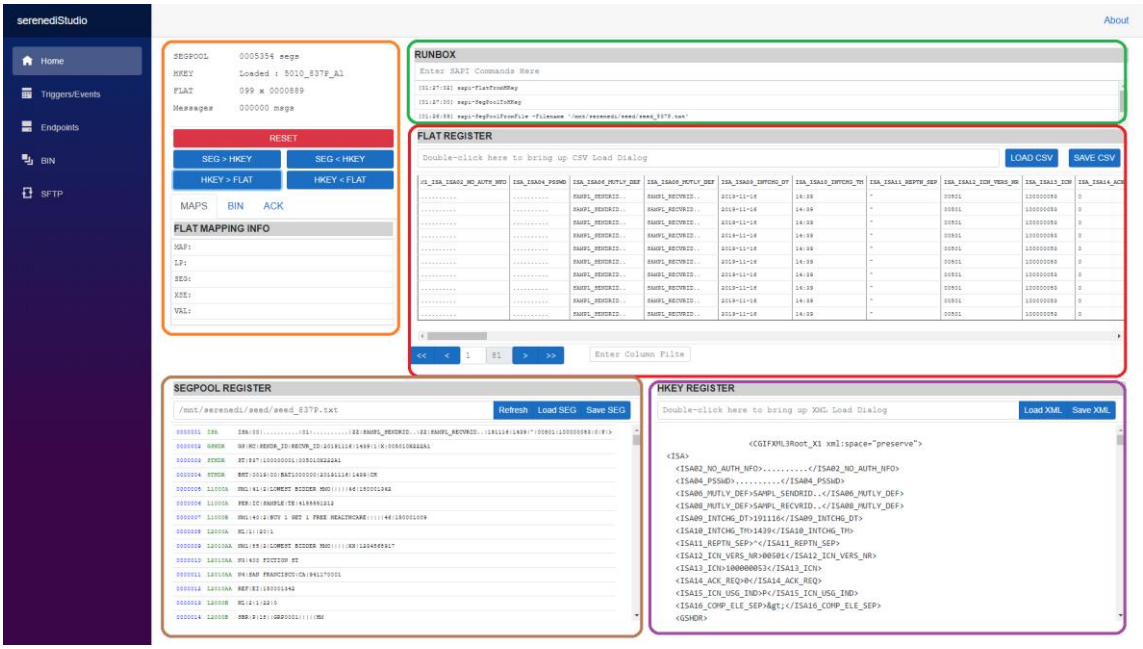
- Test the validity of EDI files and transformations
- Provide an interactive interface for testing and learning about the various registers and projections
- Show the SCORE scripting system in action so end users can learn how to script their own workflows
- Review and edit the automation triggers, and see recently triggered events and messages
- Examine the BIN system and what data is stored there
- Review and edit the SecureFTP sessions, and test folder mirror operations

SERENEDI Studio is accessible by default at http://localhost:42000 – this means it will only respond to browser requests originating from the server itself. Since SERENEDI Studio gives extensive access to the local file system, this is done for security reasons. SERENEDI Studio can be completely disabled (as shown in the installation instructions), or it can be set up to listen for external browser connections by editing the serenedi/bin/urls.txt file and setting the contents to http://0.0.0.0:42000.

Like the rest of SERENEDI, SERENEDI Studio does not need external internet access to function – the server can be completely firewalled from all internet access and it will continue to function normally.

This interface is best run at 1920 x 1080 in a web browser. It is fully tested with Chrome and the latest Microsoft Edge browser. For best results, press the Fullscreen button to remove the various browser interface elements (F11 on Chrome). The server can be accessed by multiple people at once, and each session will have its own dedicated instance of the SERENEDI engine. The triggers, endpoints, BIN system, and SFTP sessions are shared among all sessions, so care must be taken not to alter components that are being used by other people at the same time.

Main Interface

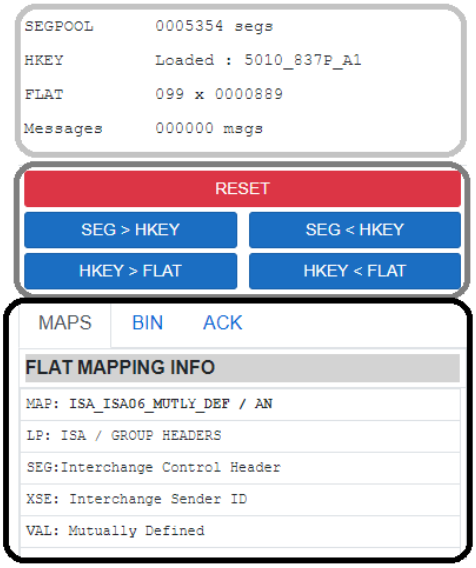


The left side of the screen is the NavBar, used to switch between various functions of SERENEDI Studio. The light orange area is the Control pane, which gives information about the current state, provides register control buttons, and gives a small tabbed interface covering additional options. The green area is the RunBox, which shows the SCORE commands generated and executed in response to all GUI activities. It also shows the results of these operations. The red area is the Flat pane, showing the current state of the Flat register. The brown area is the SegPool register, and the purple area is the HKey register.

SERENEDI Studio is oriented to operating on a single encode or decode operation at a time, and it functions to give you as much information as possible about what is occurring throughout the environment. For that reason, you need to press the RESET button every time you work on a different file or transaction.

Control Pane

The Control pane is split into three sections: the info panel (top), the register control buttons (middle), and a tab panel (bottom) that fulfills various functions.

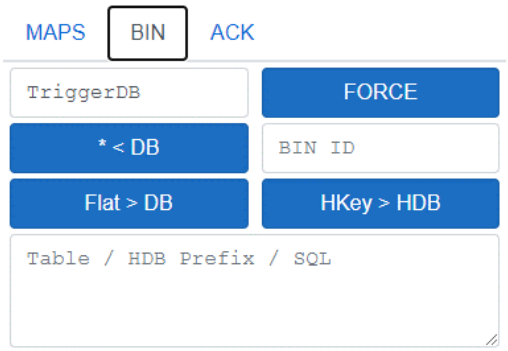


The **info panel** displays information about the current engine state: how many segments are loaded into the SegPool register, whether the HKey is loaded and with what specification of data, the number of columns and rows loaded in the Flat register, and the number of messages in the message log. If there is a critical error, the Message will highlight red.

The **register control buttons** control the major conversion functions. First, the RESET button will clear all registers and refresh all panes so the engine can be used on another major file operation. SEG > HKEY will decode the SegPool to the HKey register, while SEG < HKEY encodes the HKey to the SegPool. HKEY > FLAT projects the HKEY to the Flat register, and HKEY< FLAT projects the Flat register to the HKEY register. Errors that occur during decoding will be shown in the SegPool pane.

The **tab control** flips between three tabs. The MAPS tab accompanies the Flat pane to show more detail about the currently selected mapping and allows you to fully look up the element in the HIPAA Implementation Guides or the SERENEDI mapping documentation. If SERENEDI can "parse" the actual value underneath the cursor – for example, an NDC code or an ICD-10-CM diagnosis code – it will show that information in this box as well.

BIN Interface



The BIN tab allows you to interact with the BIN system and consists of four buttons and two text boxes.

TriggerDB is a button that switches between different database endpoints configured in the Endpoints interface. This allows you to send Flat or HDB BINs to different SQL Server or ORACLE databases. Simply clicking the button will advance through the list of defined endpoints.

The **FORCE** button is a toggle between two different modes – Force Merge and Merge. Force Merge requests the background data shuttle process to extend the schema of the destination BIN tables for any additional mappings that are not already present. Merge requests will *not* extend the schema, so additional EDI elements that are not present in the schema will be lost.

The **Retrieve BIN** (* < DB) button works in conjunction with the BIN ID textbox to the right of it and the text box at the bottom of the tab. If the BIN ID is populated, then the interface will check if that BIN ID is loaded into the environment. If it is, then it will populate either the Flat or HKEY register, depending on whether it was a Flat BIN or HDB BIN. The corresponding Flat or HKEY register panes will be regenerated as a result. If these BINs are still being populated by the background data shuttle, a modal popup will immediately appear telling you that they are still being loaded.

If the bottom **Information** text box is populated, then the Retrieve BIN will instead treat this as one of three things: a table, a SQL query or view that starts with SELECT (caps important) , or a stored procedure prefixed by the letters EXEC (caps important). It will use that as a Flat register source. If a database endpoint is supplied, SERENEDI will attempt to load the table, view, or stored procedure data from that endpoint.

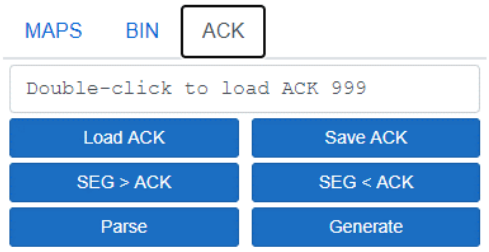
The **BIN ID** textbox starts empty. If it is populated with a valid BIN ID before the Retrieve BIN button is populated, it will drive that operation. Otherwise, it will be populated automatically by the data storage buttons below it.

The **Flat > DB** button will commit a loaded Flat register to the Flat BIN system, and the GUI will pause for a few seconds for it to get committed. The new BIN ID of the stored Flat will be shown in the BIN ID textbox immediately above. If the Information text box is populated, then that will override the default BIN table names (usually BIN_5010_837P, for example) and use the provided table name, and on the specified endpoint if that is provided. The table will be created if it does not already exist, and this BIN ID will be permanently tied to this endpoint and table name.

HKEY > HDB will commit a loaded HKEY register to the HDB BIN system, again the GUI will pause, and the BIN ID will be displayed in the text box. If the Information text box is populated, that will act as the table *prefix* for all the loop tables created during this operation, and this loop prefix and database endpoint will be permanently tied to this BIN ID.

ACK Interface

SERENEDI has the ability to automatically generate 999 Acknowledgment transactions as well as parse existing 999s. The ACK register is similar to a SegPool register in that it stores a single 999 EDI file. Although the SegPool *can* load and save 999 transactions and therefore enable very customized scenarios, the ACK register is provided as a simple facility to generate "Accept" or "Reject" transactions depending on the outcome of a SegPool decode operation.



The **ACK File Text Box** allows you to specify the path to a 999 acknowledge file. If you double-click the text box, a file dialog will pop up and allow you to load in a file that exists on the server file system, or to upload a file from your *client* file system to the remote one. Alternatively, you can supply the name of a file that you will be generating in another step.

The **Load ACK** button loads the ACK register with whatever file is selected in the ACK File text box.

The **Save ACK** button will save the ACK register (as long as it's loaded) to the file indicated in the ACK File text box.

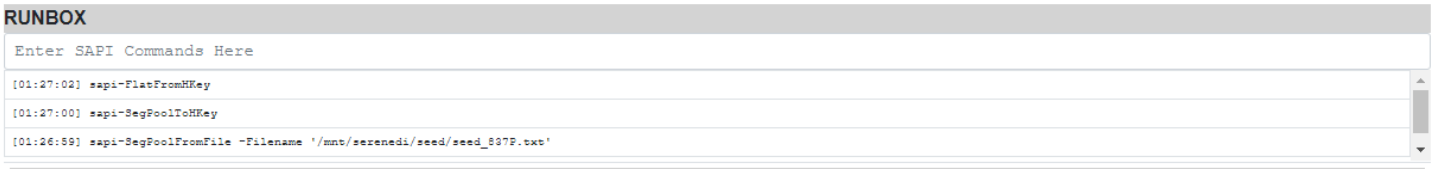
The **SEG > ACK** button allows you to pull the SegPool register to the ACK register, so long as it is a 999 file. This lets you create custom 999s user other parts of the environment, transfer the SegPool to the ACK register, reload a different SegPool, and then parse the 999 against that new SegPool and verify the results.

The **SEG < ACK** button allows you to pull the ACK register into the SegPool register. You can then commit that 999 to the BIN system or analyze it in the various Studio interfaces.

The **Parse** button will parse the 999 against the loaded SegPool register. For example, if a trading partner generates a 999 message in response to a file you sent, this operation will generate a series of messages that you could then use to analyze why the transaction was rejected, and where.

The **Generate** button will take the result of the last decode operation (a SegPool to HKey projection) and automatically generate a simple 999 transaction depending on what happened. If the decode operation was successful, it will generate a 999 showing all the transactions as Accepted. If the decode failed, it will generate an overall Rejection 999 and show exactly what line and segment resulted in the decode failure.

Runbox



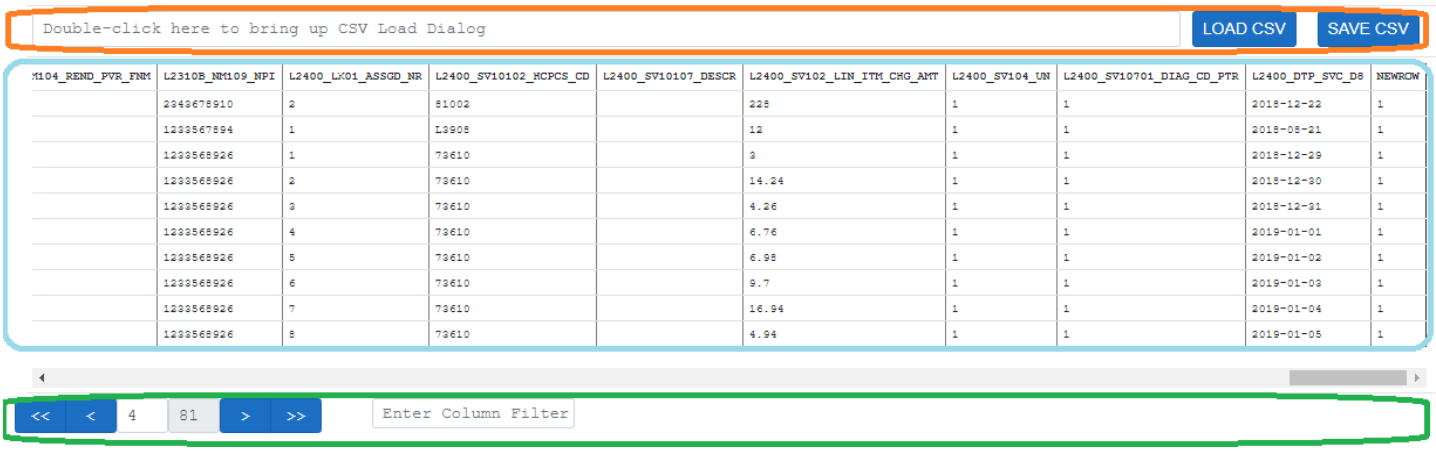
The RunBox fulfills two functions:

- Give a running display of all SERENEDI Studio operations in the form of SCORE script commands. Since all the buttons interface to the environment using these commands, it gives you a history of your activity within the interface as well as a way to learn to execute these functions through the automation system.
- Give you the ability to execute *ad-hoc* SCORE script commands.

The top line is a text box, and any commands you enter here will be executed against the current SERENEDI environment.

The next three lines are a rolling history of SCORE script commands executed by SERENEDI Studio as you run different functions in the environment. If any messages are generated during the execution, they will be displayed here as well. The messages are displayed “last first,” with the most recent action displayed on the top line.

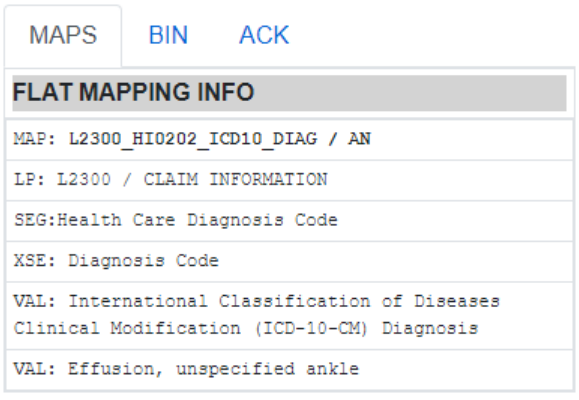
Flat Pane



At the top of the Flat pane is the **CSV interface**. If you double-click on the text box, a file dialog will pop up and allow you to load a CSV that exists on the server file system. You must click the Load CSV button to actually load it. Alternatively, if you supply a filename and click the Save CSV button, the loaded Flat register will be written to a new CSV file on the server filesystem.

The middle and largest part of the pane is the **Flat Data** control. This interface allows you to see all the data loaded into the Flat register. As there are usually dozens of data columns, the bottom scrollbar allows you to see all the data elements. The bottom **Page control** allows you to see different data pages in lieu of vertical scrolling. When you click on a cell, information about both the mapping of that cell and the data it contains is displayed in the Info Panel part of the Control Panel.

The **Column Filter** text box allows you to focus on a limited set of rows from the register. First, click on a mapping, then enter a value into the Column Filter text box. This will create a filter that displays only the rows for which the column matches the given value. Another way to do this is to double-click on any cell, and a column filter will automatically be generated against the data stored in that cell. Whenever a column filter is added, a red X is displayed to allow you to remove the last column filter added. Up to three column filters can be active at one time.



The mapping info pane to the left of the Flat pane displays information about the currently selected mapping. This makes it easier to look up in both the SERENEDI mapping documentation as well as the HIPAA implementation guides. It displays these values:

Map – This is the CGIF V2 map this column is linked to. Following this is the HIPAA EDI data type for this element: AN for string, N0 for number, DT for Date/Time, TM for Time, and R for floating-point number or money.

LP – Loop, displaying both the short name and the long name.

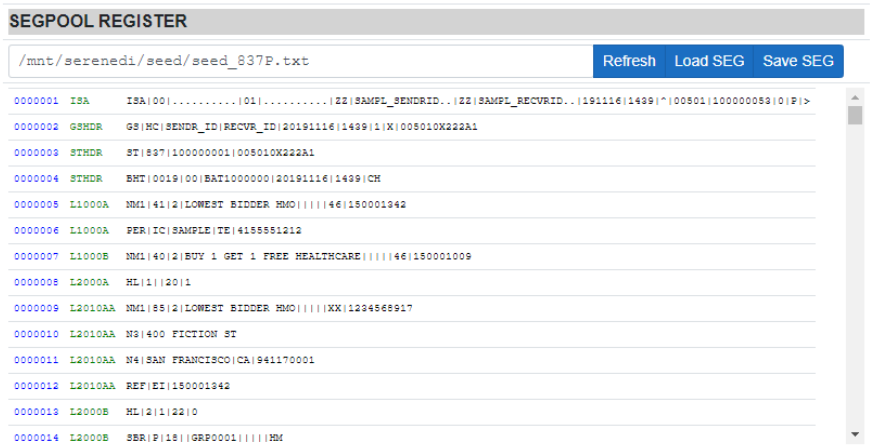
SEG – Name of the segment.

XSE – Name of the element.

SUB – (optional) Name of the composite element.

VAL – Value. This will display both the name of the value, and, if the value is given a defined name in either the implementation guides or the SERENEDI-supported code sets, that value name is given here.

SegPool Pane

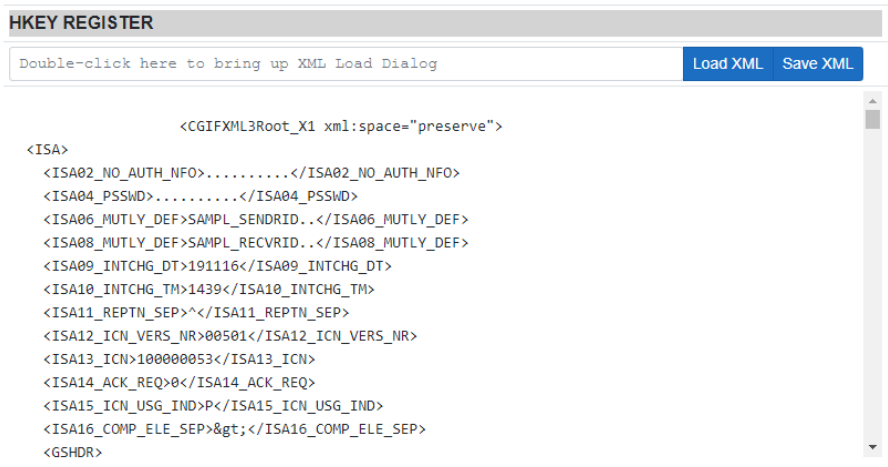


The **SegPool Pane** displays the current state of the SegPool register. If the SegPool is initially loaded, it will display the line number in blue and the segments and elements in black. If the SegPool has been decoded, additional information about the loop will be displayed in green, and any integrity errors will be displayed in red.

The top is dedicated to the **SegPool Filename** text box. Double-clicking here will pull up a file dialog; otherwise, you can enter a filename on the server filesystem. Clicking the **Load SEG** button will load the file into the SegPool register. The **Save SEG** button will save the loaded SegPool to a new file on the server file system. The **Refresh** button forces the pane to regenerate the loaded SegPool register – this is required if you enter ad-hoc SCORE commands into the RunBox that affect the SegPool register.

If the SegPool is too large to be displayed, the Load/Save functionality will still function, but the SegPool display will be disabled.

HKEY Pane



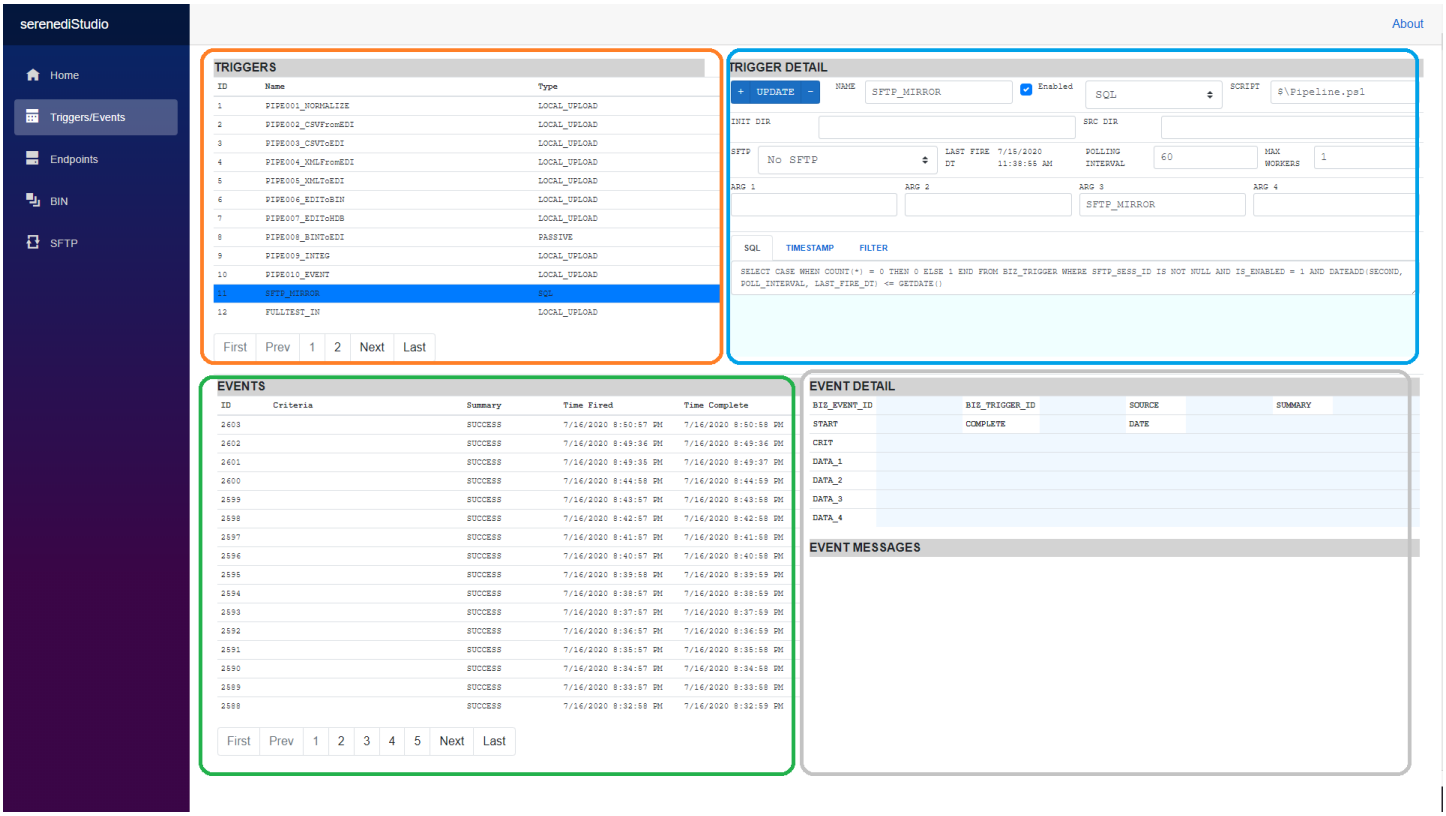
The HKey Pane displays the contents of the HKey register in an XML format. The **HKey filename** text box at the top allows you to specify a filename for the XML file; double-clicking it will bring up a Load File dialog. **Load XML** will load the HKey register with the contents of the specified XML file, and the **Save XML** button will create a new XML file based on the contents of the HKey register.

If the HKey is too large to display, the Load/Save functionality will still function, but the HKey register will not be displayed.

File Dialog

The File Dialog pops up in many cases when you choose to double-click into a file window and are given the option to select a file. Because SERENEDI is a server-based product, the File Dialog points to files and directories on the server itself and allows you to navigate them. Two of the buttons, File Upload and File Download, allow you to upload and download files from your local client.

Triggers/Events Interface

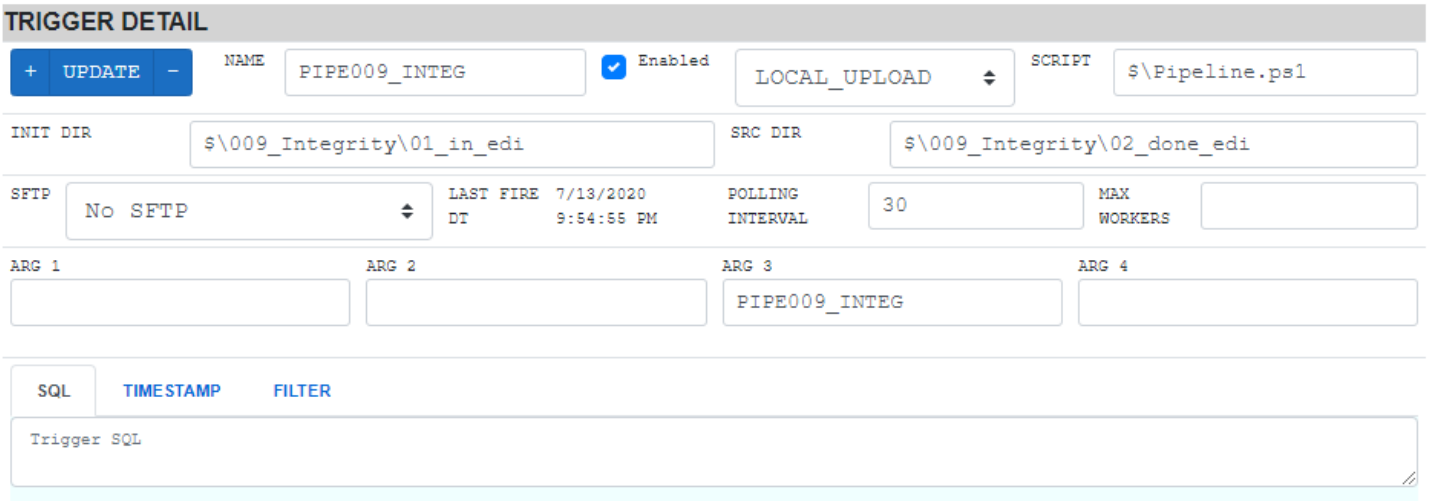


The Triggers/Events interface provides a window into the automation system and all the triggers, events, and messages. Triggers can be added, edited, and deleted, and you can view recent events as well as the messages generated during an event. This section is not meant to provide the full workings of the trigger and automation systems – instead, it defines all the controls that you as a user can manipulate, and the technical section on the automation system will go more deeply into the actual functioning of the trigger system.

The **Triggers** pane in the upper left (outlined in orange) is a view of all triggers defined in the automation system. Clicking on a trigger here will highlight it and the data for that trigger will be displayed in the **Trigger Detail** pane, outlined in blue. Events that have been fired by that trigger – sorted in time-descending order – are listed in the **Events** pane, outlined in green. Clicking on a specific event will display more detail about that event in the **Detail** pane, outlined in gray, and any messages that occurred during that event will be displayed below in the **Event Messages** pane.

Triggers

The Triggers pane allows you to view all the triggers defined in the automation system (the BIZ_TRIGGER) table. Click on a trigger to bring up the trigger editor on the right. Triggers can be sorted by ID, name, and type by clicking on the row header.



Trigger Detail

The SERENEDI automation system is highly versatile, enabling a rich set of criteria used in creating events. The Trigger Detail pane exposes these criteria to you via this interface.

The top row starts with the **Button Bar** – this control gives you the option to commit changes, add a new trigger, or delete a trigger. Triggers cannot be deleted if there are any associated events. To create a new trigger, first press the **+** button on the left of the update bar, enter the information pertaining to the trigger, and then press UPDATE. This will commit the new trigger's information to the environment, although you will have to re-select it in the Trigger interface to make further edits. Once you do, just click the Update button again, and they will be committed to the distribution database.

To make changes to the triggers, you can adjust the options presented on-screen, but they will *not* go into immediate effect. Any changes you make will only be on "in-memory" copies of the triggers, and the changes will not be committed to the database until you press the Update button.

To delete a trigger, it must not have any related BIZ_EVENT entries – these must be deleted (along with messages in the BIZ_MSG table) before the trigger can be removed. The first time you click the **-** button, a modal dialog will pop up informing you that you must press it a second time before the trigger can be deleted.

Button bars are used throughout the SERENEDI Studio interface, and they all work the same way.

The **Name** field allows you to provide a name for the trigger. The **Enabled** checkbox controls whether the trigger is active or not – the automation system will completely ignore disabled triggers for the purposes of generating new events. Next to this is the **Trigger Type** pulldown menu. It contains the following options:

PASSIVE – Passive triggers do not themselves generate events, but do allow events generated from other sources to execute the SCORE script associated with this trigger.

SQL – SQL triggers will execute queries periodically and fire triggers when the SQL evaluation equals to an integer value of 1.

LOCAL_UPLOAD – This trigger type will fire events based on files it is able to pull from the *Initial Directory* to the *Source Directory*.

LOCAL_ARCHIVE – This trigger type will fire events based on files that exist in the *Initial Directory* that have not been processed by an event before.

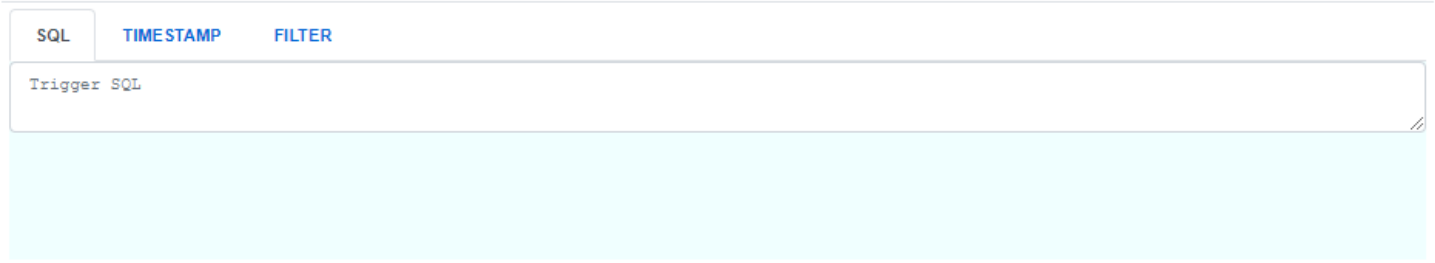
The **Script** textbox provides a location for the SCORE Script that is executed by this trigger. Instead of providing the full path, you can provide a single $ symbol to represent the full path to the pipeline directory within the environment.

The second line in the Trigger window is labeled INIT DIR. This text box is where you supply the **Initial Directory** that works in conjunction with Local Upload and Local Archive trigger types. Afterward, the SRC DIR is where you can supply the **Source Directory** for LOCAL_UPLOAD triggers.

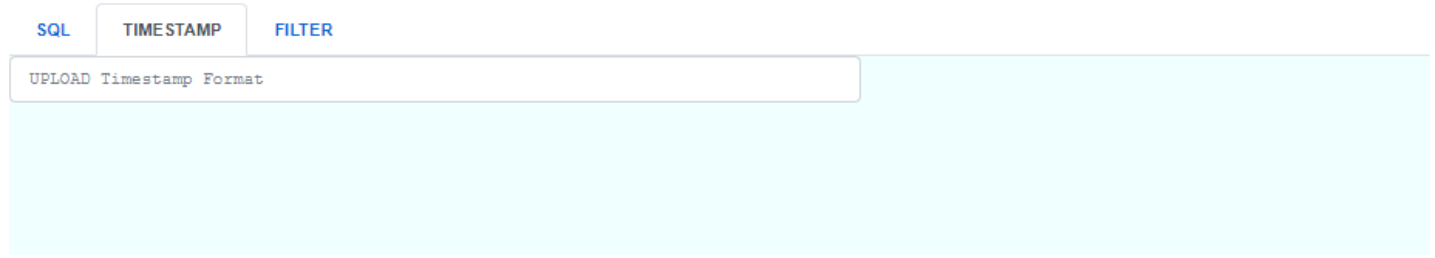
The third line allows you to supply a **Secure FTP** session linked with this trigger. Secure FTP sessions can be configured to run mirror operations, allowing you to push files to a remote server or download files. The default is NO SFTP. After this is a date/time stamp of when the trigger was last fired. Then there is a numeric text box for the **Polling Interval**, which indicates the number of seconds between polling of the trigger conditions. It's followed by the **Maximum Workers** textbox, which allows you to set limits on the number of worker processes SERENEDI will commit to this trigger's events at any one time.

The fourth line defined **Argument 1**, **Argument 2**, **Argument 3**, and **Argument 4**. These will automatically populate event data when this trigger is fired and enable a single SCORE script to service multiple triggers.

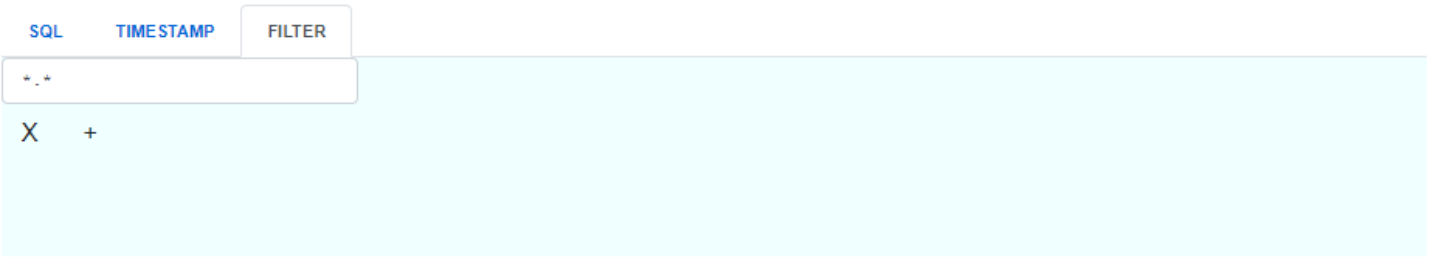
Below the main body of the Trigger definition is a tab control that regulates the auxiliary behavior of the trigger.



The **SQL Tab** is used in conjunction with SQL triggers. The above example shows the SQL used to trigger the SFTP_POLL pipeline that runs mirror operations on triggers with the SFTP session set, and is due for a poll operation.



The **Timestamp Tab** is used in conjunction with LOCAL_UPLOAD triggers and allows you to set timestamps within the files as they are moved from the Initial Directory to the Source Directory. This is useful if you have a trading partner that sends files with a fixed name and you need to differentiate it from the other files using a timestamp.



The **Filter Tab** is a way to use wildcards like ? and * to filter the files that trigger events. If more than one set file filter is defined, then the trigger will enforce *file sets* – such as a CSV header file and CSV detail file. This could be set up with one filter as *HDR.CSV and another set as *DTL.CSV. For example, suppose the event will only be generated if the wildcards match on two files: one file ending with HDR.CSV, and a second file ending in DTL.CSV. The actual event will be linked only to the file that fulfills the first-listed filter. Generally when two or more filters are established, these are primarily used with flat-format files and not X12 EDI files, which are self-contained.

The X button allows you to delete the last filter defined, and the **+** button allows you to define a new file filter.

Events

EVENTS

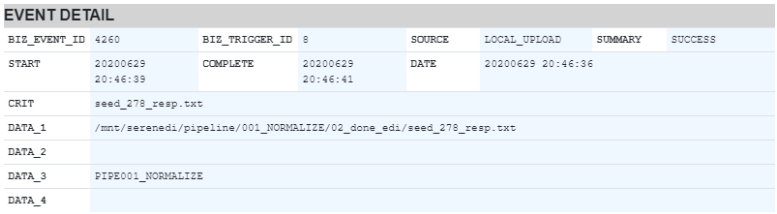
ID	Criteria	Summary	Time Fired	Time Complete
4260	seed_278_resp.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:41 PM
4261	seed_824.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:41 PM
4259	seed_278_req.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:38 PM
4258	seed_820.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:38 PM
4255	seed_820X.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:38 PM
4256	seed_271.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:38 PM
4257	seed_270.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:38 PM
4254	seed_277CA.txt	SUCCESS	6/29/2020 8:46:36 PM	6/29/2020 8:46:37 PM
4252	seed_276.txt	SUCCESS	6/29/2020 8:46:35 PM	6/29/2020 8:46:37 PM
4253	seed_834.txt	SUCCESS	6/29/2020 8:46:35 PM	6/29/2020 8:46:37 PM
4251	seed_277.txt	SUCCESS	6/29/2020 8:46:35 PM	6/29/2020 8:46:37 PM
4249	seed_835.txt	SUCCESS	6/29/2020 8:46:35 PM	6/29/2020 8:46:37 PM

First Prev 1 2 Next Last

This shows the most recent 5,000 events in the automation system. Although it's a mostly non-interactive view, by clicking on the ID, Criteria, Summary, Time Fired, and Time Complete column headers, you can sort the results. The bottom represents the pager to switch through multiple pages of events. Selecting an event row will bring up the detail in the Event Detail pane, and also any messages associated with this event in the Event Messages pane.

Event Detail

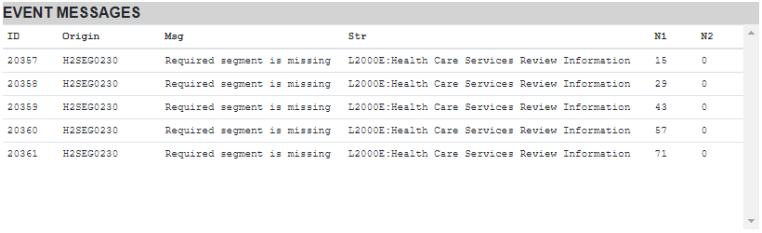
This pane shows all the details associated with a specific event, including all parameters, summary information, the times it was started and completed, and when the event was actually generated.



EVENT DETAIL

BIZ_EVENT_ID	4260	BIZ_TRIGGER_ID	8	SOURCE	LOCAL_UPLOAD	SUMMARY	SUCCESS
START	20200629 20:46:39	COMPLETE	20200629 20:46:41	DATE	20200629 20:46:36		
CRIT	seed_278_resp.txt						
DATA_1	/mnt/serenedi/pipeline/001_NORMALIZE/02_done_edi/seed_278_resp.txt						
DATA_2							
DATA_3	PIPE001_NORMALIZE						
DATA_4							

Event Messages

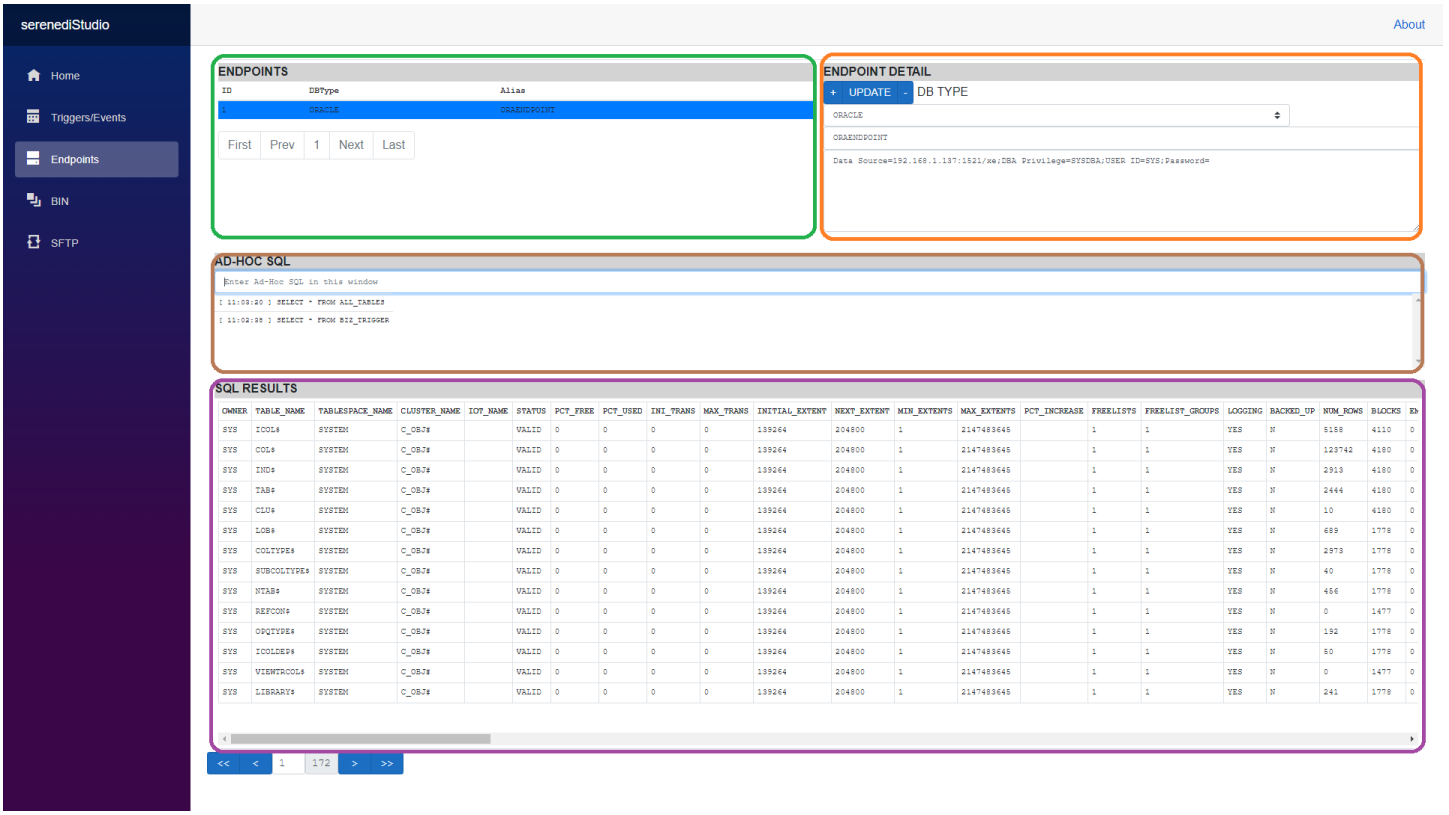


EVENT MESSAGES

ID	Origin	Msg	Str	N1	N2
20357	H2SEG0230	Required segment is missing	L2000E:Health Care Services Review Information	15	0
20358	H2SEG0230	Required segment is missing	L2000E:Health Care Services Review Information	29	0
20359	H2SEG0230	Required segment is missing	L2000E:Health Care Services Review Information	43	0
20360	H2SEG0230	Required segment is missing	L2000E:Health Care Services Review Information	57	0
20361	H2SEG0230	Required segment is missing	L2000E:Health Care Services Review Information	71	0

This pane shows the messages associated with an event in numerical order.

Endpoints Interface



The Endpoints interface enables you to create, test and delete database endpoint connections. If you'd like to send BIN table data to databases other than the serenediCore distribution database, you must define the information about it here first.

Endpoints

The **Endpoints** pane outlined in green in the upper left is a list of all defined endpoints. Clicking on the column headers allows you to sort the results. Clicking on a database endpoint will bring up its information in the Endpoint Detail pane on the right.

Endpoint Detail

The orange-outlined **Endpoint Detail** pane gives you the ability to create, update, and delete database endpoints and set the connection string and database type for the endpoint. The **+** button will create a blank endpoint in the window, but will not actually commit it to the environment until you enter the information and press the Update button. To make a change to the endpoint, highlight it, make the change, and then click Update. To delete an endpoint, the delete button, -, needs to be hit twice to prevent accidental erasures.

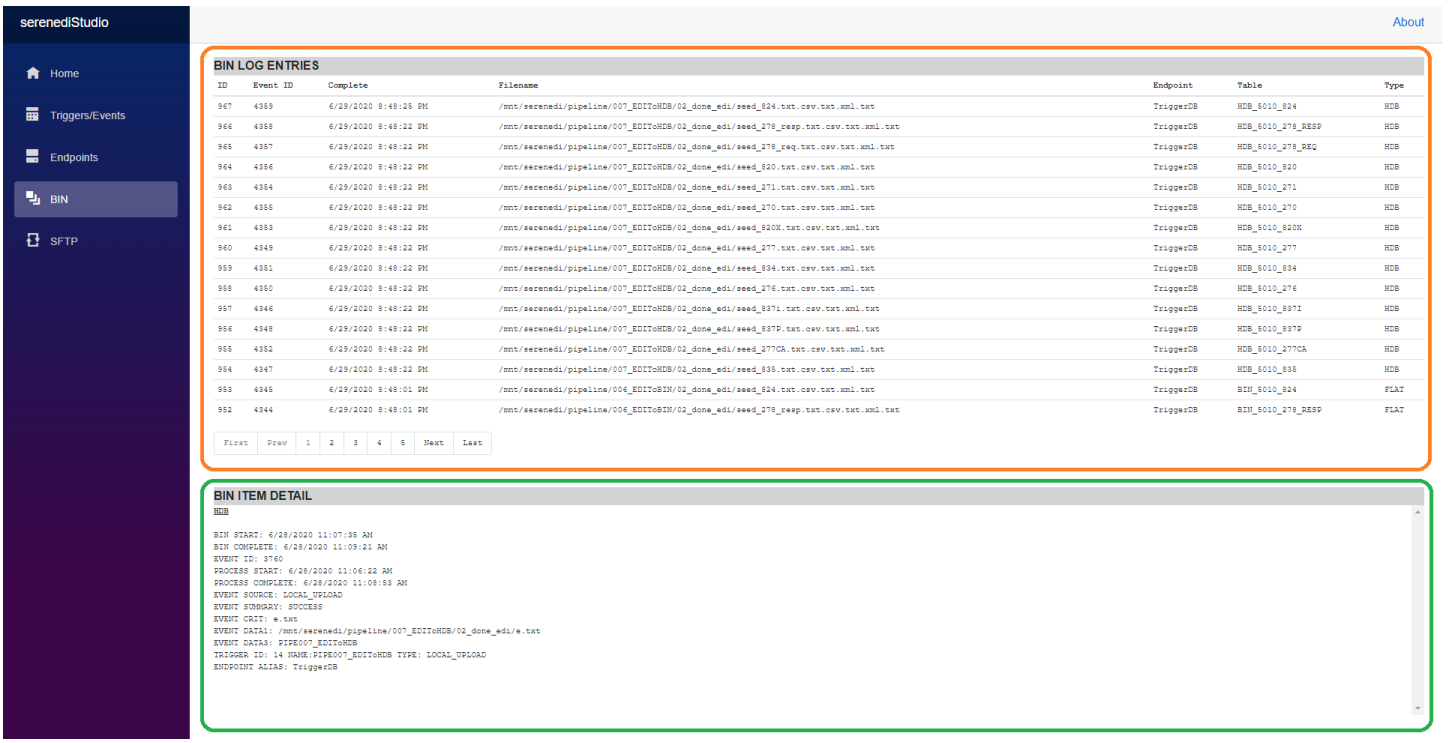
Ad-Hoc SQL

The **Ad-Hoc SQL** pane outlined in brown gives you the ability to run arbitrary SQL against that connection – or against the serenediCore database if no database is selected. Any type of SQL can be run – updates, selects, or stored procedures, and if there are results, they'll be displayed below.

SQL Results

The **SQL Results** pane shows you the results of that SQL. There is no row limiter in this window, so care should be taken not to enter SQL that overwhelms both the user interface and the server itself.

BIN Interface

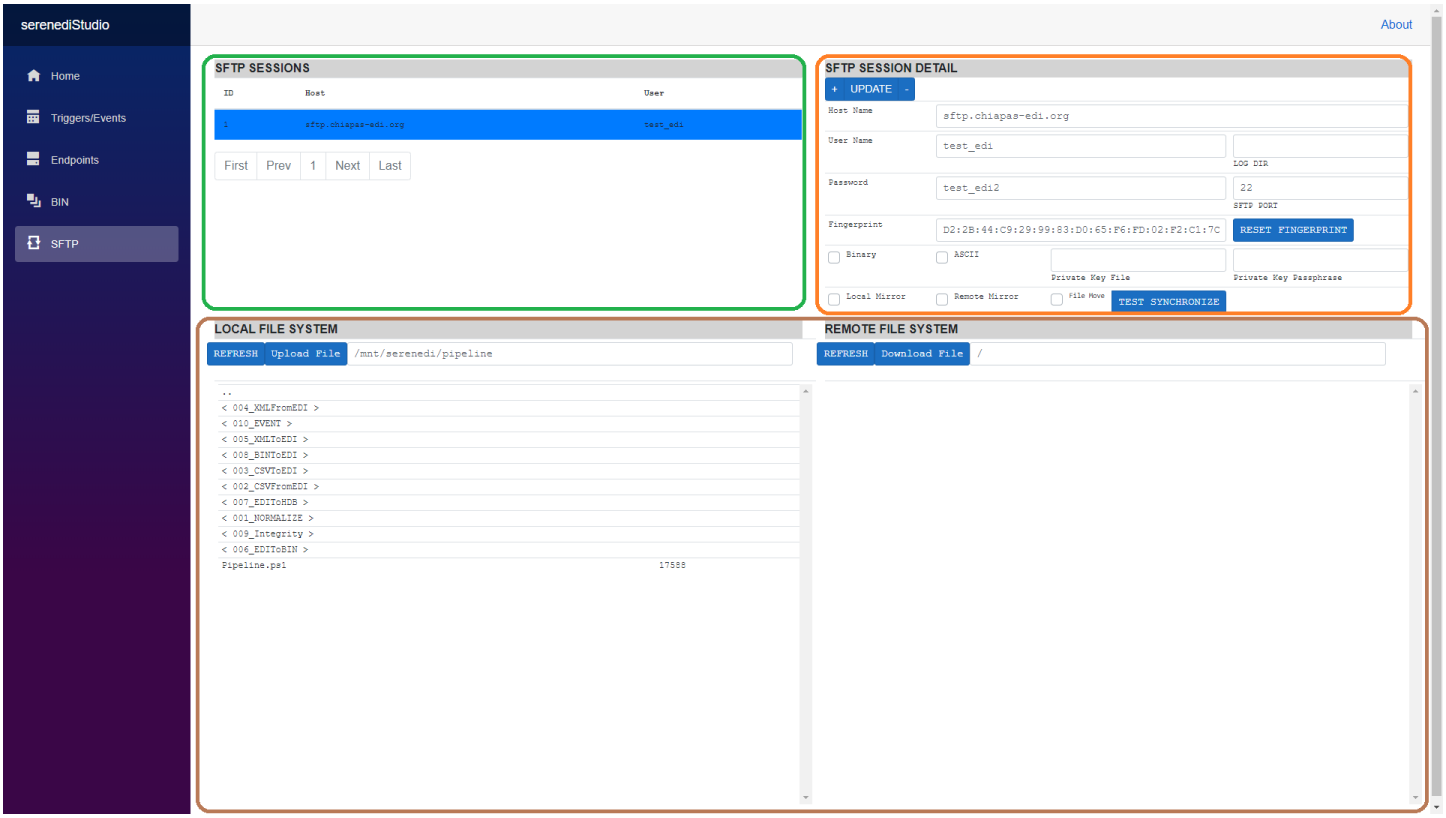


The BIN interface provides a window into the BIN system, which stores six types of information: SegPool, XML, HDB, flat, compressed binary, and binary data. The HDB and flat data types are stored in human-accessible SQL DataTables, and can be stored on other databases; the other four data types are stored in the BIN_BLOB table that must reside in serenediCore, the distribution database.

The **BIN LOG Entries** window displays the source filename, the timestamp when the BIN was completed, the table and endpoint where the BIN data is stored, and the type. Only the most recent 5,000 BIN items are displayed. The results can be sorted by clicking on the column headers.

BIN Item Detail gives more detail about both the BIN item and the event that created it.

SFTP Interface



SERENEDI incorporates a number of SFTP commands into the SCORE scripting language, and also provides a way to enable local and remote directory mirroring by using special SFTP mirroring triggers. The SFTP Pipeline automatically polls these SFTP triggers that are linked to SFTP sessions. This interface allows you to test the SFTP sessions directly, set options for logging and mirroring, and test various operations.

The **SFTP Sessions** pane outlined in green is a list of all defined SFTP sessions. After you select one of the sessions, its details are set in the **SFTP Session Detail** pane. Below this, outlined in brown, are the **Local File System** and **Remote File System** panes.

SFTP Sessions

This pane will simply list the SFTP Session ID, the host name, and the user for the session. Sorting by these fields can be done normally by clicking on the column headers.

SFTP Session Detail

At the top is the **Button Bar** control that handles creating new SFTP sessions, updating existing sessions, and deleting existing sessions. To create a new session, click **+**, enter the information for the session, then click Update. Changes to the session will not be committed to the database until the Update button is clicked. Likewise, to delete a session, you must click the **-** button twice.

Below this is the **Host Name** text box. You may use an IP address or a fully qualified host name. Below that is the **User Name** and **Password** that will be used to log into the specified SFTP server. Alongside these text boxes, the **Log Dir** will enable you to provide a directory to place logging files. The **SFTP Port** enables you to provide an override for the SFTP port for connections.

Below that is the **Fingerprint**, used to validate the SFTP server's identity. Initially it is set to a null value, and it will be set on the first operation or Update operation. If the server changes for any reason, you will need to press the **RESET Fingerprint** button to clear it again, or the server session will not connect.

Below this are options to force **Binary** and **ASCII** file transfers. The **Private Key File** will reference a file on the local system that stores the public and private keys used for the session, and the **Passphrase** provides a way to unlock the Private Key file if it is locked.

The **Local Mirror** checkbox will direct the SFTP session to download new files from the remote server to the local file system. The **Remote Mirror** checkbox will direct the SFTP session to upload new files from the local file system to the remote server. The **File Move** checkbox directs the SFTP session to delete the file from the local or remote file system once the file has been downloaded or uploaded.

The **Test Synchronize** button will execute a mirror operation using the Local File System and Remote File System values.

Local File System / Remote File System

These two panes allow you to experiment with uploading and downloading files, viewing the files on the remote server, and set things up so you can click the **Test Synchronize** button to see how the mirror operation functions with the parameters set for this session. Initially, you'll need to click the **Refresh** button to get the active directory on both the local and remote file systems. Click on the directory folders to move to different locations. You can also change the path directly in the Local Path / Remote Path text boxes and click Refresh to go immediately to a specific directory in the local or remote file systems.

INTRODUCTION

A lot of the learning curve of using SERENEDI involves understanding how the integration platform maps EDI transaction elements to database elements. This process is just as complex as the hierarchical HIPAA implementation guides themselves.

Before going into detail about the CGIF2 mapping naming conventions, it's important to note the underlying business objectives behind this system. HIPAA-compliant EDI transactions are laid out in a completely hierarchical fashion – starting with outer envelope segments such as ISA, GS, and ST, then wrapping up with the closure envelope segments of SE, GE, and IEA. Within this structure, data is arranged in loops, segments, and elements. A loop is an aggregate of segments, whereas a segment is a collection of elements. Elements and composite elements are generally tied to specific business items such as claim numbers, last names, and the many thousands of other items that are discretely identified within the HIPAA implementation guides. These elements are surrounded by a discretely defined scaffolding that positions the data elements correctly within the hierarchy.

SERENEDI's architecture and CGIF2 mapping system are oriented to enable end-users to focus as much as possible on the business information contained within the transaction and ignore the scaffolding that contains it. At the same time, this mapping system needs to completely encapsulate every element within a transaction. Because the HIGs themselves can sometimes allow for a deeply complex nesting of iterated loops and segments, the mapping system must account for every single possibility.

One thing to note is that the CGIF2 mappings *completely* define the contents of an EDI file. The SERENEDI engine is programmed completely through the presence and contents of these mappings. It differs from many EDI packages that contain XML schemas and enable end-users to alter them to fit a particular business scenario. Because the mapping system is tightly integrated to the specifications, it does not allow for "custom" segments or departures from the HIPAA implementation guides.

Therefore, the simplest way to define CGIF2 is as a way of encapsulating all the complexity of locating a single data element within the HIPAA implementation guide and projecting it into a two-dimensional space. In this way, the hierarchy's many complexities are "unrolled" and flattened to a two-dimensional table, as that is the natural data structure for an enterprise system: a relational database. Since this format rigidly follows all the complexities of the HIGs, then it must stand to reason that any valid HIPAA EDI transaction can be projected into a flat database table. It therefore holds that to *create* a new HIPAA EDI transaction, the user must generate an SQL data table that stages the data exactly as if the file were freshly decoded in SERENEDI, with field names that strongly adhere to the CGIF2 conventions.

In order to make this mapping convention work, SERENEDI categorizes all the loops into a number of specific categories. As defined within the HIGs, loops often have specific relationships with other loops, and these relationships have to be a part of the mapping methodology in order to pinpoint an element's exact location within the hierarchy.

The following sections explain how SERENEDI maps elements and binds them to the hierarchy. All these maps are provided in the HTML files inside of the serenedi/docs/specs directory. As per the 5010_837I.html file:

L2310F - REFERRING PROVIDER NAME

L2310F	NM1	Referring Provider Name		
03		L2310F_NM103_REF_PVR_LNM	String	Referring Provider Last Name
04		L2310F_NM104_REF_PVR_FNM	String	Referring Provider First Name
05		L2310F_NM105_REF_PVR_MNM	String	Referring Provider Middle Name or Initial
07		L2310F_NM107_REF_PVR_SFX	String	Referring Provider Name Suffix
09		L2310F_NM109_REF_PVR_ID	String	Referring Provider Identifier
L2310F	REF	Referring Provider Secondary Identification		
02		L2310F_REF_STAT_LIC_NR	String	State License Number
02		L2310F_REF_UPIN	String	Provider UPIN Number
02		L2310F_REF_PVR_COMM_NR	String	Provider Commercial Number

L2320 - OTHER SUBSCRIBER INFORMATION (Single Iteration)

L2320	SBR	Other Subscriber Information		
01		L2320_xx_SBR01_PYR_RESP_SEQ_NR	String	Payer Responsibility Sequence Number Code
02		L2320_xx_SBR02_IND_RELAT_CD	String	Individual Relationship Code
03		L2320_xx_SBR03_INS_GRP_PLCY_NR	String	Insured Group or Policy Number
04		L2320_xx_SBR04_OINS_GRP_NM	String	Other Insured Group Name
09		L2320_xx_SBR09_CLM_FIL_IND_CD	String	Claim Filing Indicator Code

The specification code and type are found at the top of the file and are part of the filename. The loop short and long names are listed in bold here. Below that, you will find a list of segments and the actual maps. The Element Index is the first column. If there is a *composite element index*, it will be listed in the second column in red. The actual map is the third column. In some cases, you'll see different-colored characters within the map itself, indicates variation according to the situation. In the above example, the loop is categorized as a *Single Iteration*, which means that the Other Subscriber Information can repeat a certain number of times. The green *xx* identifies the number of the iteration, starting at 01. This 01 is carried into all *child loop maps*, which is how all the information in those maps is related together with this specific iteration of the 2320 loop.

CGIF3 Loop Types

STANDARD – This loop can iterate one time or many times, and has no special relationship with the parent or child loops. It's the designation for all loops within a specification's *main data encoding branch*, the set of parent/child loops encoding the information that is the general purpose of the transaction.

SINGLE ITERATION – This loop is defined in the HIG as not being within the main data encoding branch and having a specific number of repeats. In general, these loops encode auxiliary information. A good example is Loop 2320 in both the 837 Institutional and 837 Professional implementation guides, which establishes Coordination of Benefits information associated with a claim. The loops in this example can repeat up to 10 times to relay information for 10 different COB providers. Every mapping in a Single Iteration loop must contain a number that indicates a specific iteration of this loop.

INHERITED ITERATION – This loop is a child of the Single Iteration loop described above. Every mapping for this loop has to *inherit* the iteration of the parent loop. An example of this is the 837 Institutional loop 2330D, Other Payer Operating Physician. Therefore, any mappings of 2330D will be associated with specific iterations of the parent 2320 loop.

INHERITED ITERATION & VALUE – This loop is just like the Inherited Iteration loop type described above, but with the added twist of multiple qualifiers present within the header segment. One example is Loop 2330C in the 837 Professional HIG. This loop iterates along with the parent 2320 loop, but also with the qualifier present in the NM1 segment, determining whether that loop represents a COB Referring Provider or a COB Primary Care Provider.

QUALIFIED VALUE – This represents a loop that contains multiple qualifiers in the header segment that change the information composition of all mappings within that loop. Generally, the first element in the first segment determines the most pertinent information about the loop. One example is the 837 P 2420F loop, Referring Provider. Each of the two possible iterations can encode information about either a Referring Provider or a Primary Care Provider.

INHERITED VALUE – If a Qualified Value loop has child loops, they inherit the value of the parent loop within the mapping. This only occurs in a few instances, but one example is the Transmission Receipt Control Identifier loop 2200A, which is a child of the 2100A Information Source Name loop within the 277CA specification.

CUTOUT – This is an *artificially created* loop that is neither present nor defined within the HIPAA Implementation Guides. Instead, it is a SERENEDI convention to "cut out" a segment with unbound repetitions from the loop and isolate that segment within its own child loop. SERENEDI will name cutout loops after the original loop along with an "X" or "Y" suffix to differentiate these from more conventional loops. This gives SERENEDI the ability to encode any number of repetitions of this segment without needing to define additional mappings for the different iterations. In the FLAT encoding system, these loops are used in conjunction with the mandatory NEWROW column to call out rows that are dedicated to encoding cutout loop iterations – if NEWROW is set to 0, then all loops *except* cutouts are ignored.

In general, when cutout mappings have data, they will be encoded to a new segment in all cases except for the FLAT encoder when the cutout loop parent and the parent of that loop both have a valid iteration repeat set to 1. In that case, data will be encoded only if there is a delta detected within the cutout or parent loops, or if NEWROW is set to 0. This scenario is present with the 277CA where the STC cutout is embedded within the parent 2200B loop (1 valid iteration) and it's parent 2000B loop (1 valid iteration), and allows the cutout information to be encoded in a more natural way.

Element Mapping

Elements in SERENEDI have four discrete data types: String, Integer, Floating Point, and Date/Time. For maps that involve date ranges (which are prefixed by an RD8 qualifier), there are actually *two* mappings that correspond to the beginning and ending date range, which are suffixed with RD8_1 and RD8_2. These data types are especially important when working with database tables, as they reflect how the cells are stored and queried.

Element maps have the following components: the Segment Code and Element Abbreviation or Qualifier are always required; the other components may or may not be present, depending on the mapping.

Segment Iteration	Segment Code	Segment Suffix	Element Index	Composite Element Index	Element Repeat Index	Element Abbreviation or Qualifier

Segment Iteration (OPTIONAL)

For segments with a fixed number of repetitions, the maps enable binding to a specific segment iteration by prefixing it with a two-digit number. This number is at least 02 or above – the first iteration of any segment does not need any prefix.

Segment Code (MANDATORY)

This is the two- to three-digit code of the segment itself, like REF or CLM.

Segment Suffix (OPTIONAL)

Sometimes the combination of Element Abbreviation and Segment Code is not sufficient to uniquely identify a segment, especially when the specification calls for a run of segments describing closely aligned information, like in the 837 guides with HI segments. For these cases, a single character suffix is added to the segment.

Element Index (OPTIONAL)

This is a two-digit index referring to the exact element of the map. It occurs for most mappings, but may be omitted for segments that do not convey many mappings, such as DTP and REF segments.

Composite Element Index (OPTIONAL)

This is a two-digit index referring to the index within a composite element.

Element Repeat Index (OPTIONAL)

In a few elements among the many in the HIG, sometimes an element is able to *repeat*. For example, the *composite race or ethnicity information* element in the 834 DMG04 can repeat 10 times, separated by the element repeat character specified in the outer ISA segment. In this mapping system, an E followed by a two-digit number allows these elements to be mapped.

Element Abbreviation or Qualifier (MANDATORY)

This specifies additional information about the element being mapped and is usually a compressed shorthand for the element's Implementation Name given in the HIGs. If the element is the identifier of a qualifier/identifier pair (where both elements are listed in the HIG as REQUIRED), then this abbreviation will relate to one of the qualifiers in the preceding element. For these cases, it means that a single mapping is bound to two elements, both the qualifier and identifier, in the target EDI file, and also that it is not necessary to know what the qualifier is, just what the data point represents.

Note that Date/Time elements have special suffixes that bind the information to a certain format in the EDI file. These possible valid suffixes for every element will be provided in the associated mapping documentation, but they will be a part of this set:

TM – Four-digit timestamp, HHMM.

TM6 – Six-digit timestamp, HHMMSS.

TM8 – Eight-digit timestamp, HHMMSScc.

DT – For six-character date fields, this will be a six-character date following YYMMDD. For all others, it will be a 12-digit date time stamp following YYYYMMDDHHMM.

D8 – This is a normal eight-digit date, following YYYYMMDD.

RD8_1, RD8_2 – These suffixes denote the lower and higher dates of a date time span for a single element. Date Range elements are always "split" into two columns for the lower and upper part of the date span.

EXAMPLES

A CGIF2 map is subdivided into three sections. The total amount of characters for each map will never exceed 30 characters – this helps keep the length of the different mappings manageable and maintains compatibility with legacy Oracle database systems. Here are some examples of the different conventions used in CGIF2 mappings:

Example #	Loop Identifier & Qualifier	Segment/Element Map	Full Mapping
1	ISA	ISA02_NO_AUTH_NFO	W2_ISA_ISA02_NO_AUTH_NFO
2	L2300	CLM02_TOT_CLM_CHG_AMT	L2300_CLM02_TOT_CLM_CHG_AMT
3	L2320_02	CAS03_ADJ_AMT	L2320_02_CAS03_ADJ_AMT
4	L2200DX	STC04_TOT_CLM_CHG_AMT	L2200DX_STC04_TOT_CLM_CHG_AMT
5	L2100A_IL	DMG0501_E03_RAC_ETH_CD	L2100A_IL_DMG0501_E03_RAC_ETH_CD
6	L2300	DTP_STMNT_RD8_2	L2300_DTP_STMNT_RD8_2
7	L2330C_02P3	REF_PVR_COMM_NR	L2330C_02P3_REF_PVR_COMM_NR
8	STHDRX	PLB030_PVR_ADJ_ID	STHDRX_PLB030_PVR_ADJ_ID

Example 1

W2_ISA_ISA02_NO_AUTH_NFO

If we look at the 837 Institutional HIG C.1 section on control segments, we'll see the definition of the ISA segment, and by the mapping conventions established above, we know that this is a map to the ISA loop, the ISA segment, Element 02. One unusual thing is the W2 prefix, which occurs once and only once in a set of mappings, and is what tells the SERENEDI encoding engine exactly what specification these maps belong to. This is universal to all valid sets of CGIF2 maps, that the very first (and *only* the first) mapping needs to establish the specification being mapped.

The NO_AUTH_NFO element suffix is referencing the ISA Element 01 qualifier 00, "No authorization information present." Therefore, it maps to two discrete elements in the ISA segment, ISA01 with the qualifier set to 00, and ISA02, where the mapped data is actually stored.

Example 2

`L2300_CLM02_TOT_CLM_CHG_AMT`

To see more about this mapping, we can look it up in the internal mapping documentation, located at:

C:\serenedi\docs\specs\ 5010_837I_A2.html / Loop 2300:

L2300 - CLAIM INFORMATION

L2300	CLM	Claim Information		
01		L2300_CLM01_PT_CTL_NR	String	Patient Control Number
02		L2300_CLM02_TOT_CLM_CHG_AMT	Decimal	Total Claim Charge Amount

The data type is present in the fourth column, meaning that the information being stored in this Total Claim Charge Amount field is **floating point**. In SQL Server, this is equivalent to the FLOAT(53) data type.

Example 3

`L2320_02_CAS03_ADJ_AMT`

This is an example of an Adjustment Amount segment.

C:/serenedi/docs/specs/5010_837I_A2.html / Loop 2320:

`L2320_02_CAS03_ADJ_AMT`

02		L2320_xx_nnCAS02_ADJ_RSN_CD	String	Adjustment Reason Code
03		L2320_xx_nnCAS03_ADJ_AMT	Decimal	Adjustment Amount
04		L2320_xx_nnCAS04_ADJ_QTY	Decimal	Adjustment Quantity

In the above listing, the green xx stands in for the numeric 2320 loop iteration, and the red nn is for the segment iteration prefix, which is only present on the second iteration and above.

If it were necessary to send a second CAS segment right after the first one, the mapping would gain a segment iteration index and look like this:

`L2320_02_`**`02`**`CAS03_ADJ_AMT`

Example 4

`L2200DX_STC04_TOT_CLM_CHG_AMT`

This is a 277CA map, found here:

C:/serenedi/docs/specs/5010_277CA.html / Loop 2200DX:

L2200DX - CLAIM STATUS TRACKING NUMBER - STC CUTOUT

L2200DX	STC	Claim Level Status Information		
01	01	L2200DX_STC0101_HTCRCLM_CAT_CD	String	Health Care Claim Status Category Code
01	02	L2200DX_STC0102_HTCRCLM_STATCD	String	Health Care Claim Status Code
01	03	L2200DX_STC0103_ENTY_ID_CD	String	Entity Identifier Code
02		L2200DX_STC02_STMT_INFO_EFF_D8	Date/Time	Status Information Effective Date
03		L2200DX_STC03_ACTN_CD	String	Action Code
04		L2200DX_STC04_TOT_CLM_CHG_AMT	Decimal	Total Claim Charge Amount

This is an example of a *cutout* map. As defined above, the 2200DX loop does not appear in the HIPAA implementation guides; instead, it's a convention of *cutting out* segments that have infinite repetitions so they can be mapped in a different way. Cutout maps are mapped *vertically* across multiple database rows. If a cutout iterates more than once, then a new database row that copies all data values up to the L2200D loop is presented, but with new values for the L2200DX maps. To prevent SERENEDI from seeing the L2200D values as a new row, the mandatory field NEWROW is set to 0, which blocks out all other columns from the data parser and focuses only on cutout mappings.

Normally, NEWROW is set to 1 and is mandatory on every CGIF2 Flat, even for specifications that lack cutout loops. When NEWROW is 1, then the data parser is guaranteed to emit a new row for the deepest Standard loop present in the data row. When NEWROW is 0, then all other mapped values are ignored and only the cutouts are focused on. The other values may be present, especially in SQL Views, to maintain the sequence in properly sorted order, but the parser will scan each row for non-null values in cutout mappings.

Here's an example of how this works:

L2200DX_STC04_TOT_CLM_CHG_AMT	NEWROW
100.00	1
110.00	0
120.00	0
130.00	0


```
STC*A6:123:00*20190101*WQ*100.00~
STC*A6:123:00*20190101*WQ*110.00~
STC*A6:123:00*20190101*WQ*120.00~
STC*A6:123:00*20190101*WQ*130.00~
```

Note, that it may give rise to a question: what if there are multiple cutouts at multiple levels? The data is presented in the same way, and the parser will intelligently restructure the database output and link them into the correct places in the EDI hierarchy. For example, earlier in the hierarchy is the loop 2200C and the associated cutout, 2200CX. This gives an example of how multiple levels of cutouts are presented and encoded in the destination EDI.

L2200CX_STC04_TOT_SBR_CHG_WRK	L2200DX_STC04_TOT_CLM_CHG_AMT	NEWROW
210.00	100.00	1
250.00	110.00	0
	120.00	0
	130.00	0


```
…
STC*A1:19*20190101*WQ*210~
STC*A1:19*20190101*U*250~
HL*4*3*PT~
NM1*QC*SMITH*JOE***MI*123456ABC~
TRN*2*JOESMITH12345~
STC*A6:123:00*20190101*WQ*100.00~
STC*A6:123:00*20190101*WQ*110.00~
STC*A6:123:00*20190101*U*120.00~
STC*A6:123:00*20190101*U*130.00~
```

Example 5

`L2100A_IL_DMG0501_E03_RAC_ETH_CD`

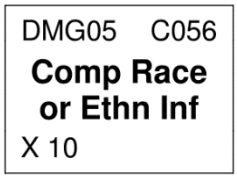
This example is drawn from the 834 Implementation Guide. The SERENEDI maps are found here:

C:\serenedi\docs\specs\5010_834_A1.html / Loop 2100A:

05	01	L2100A_yy_DMG0501_Enn_RAC_ETH_CD	String	Race or Ethnicity Code
05	03	L2100A_yy_DMG0503_Enn_CS_RC_ETH	String	Classification of Race or Ethnicity

It demonstrates a fairly complex mapping – let's start at the top:

1. **Value Qualified Loop** – The red yy in the mapping guide indicates that one of the two Loop Qualifier values should be placed here. For the 834 2100A loop, these are **70** to indicate a Corrected Insured loop, or **IL** to indicate an Insured Member loop.
2. **Composite Element** – These values represent composite elements, which are elements defined in the HIGs that are nested within other elements. From the parsing of the DMG0501 part of the map, this means the first composite element within the fifth element of the DMG segment.
3. **Repeated Element** – Looking at the element definition for this element within the HIG, we'll see this box:

   The X 10 means that this element – meaning all of the composite elements – can repeat up to 10 times. The **E03** provided in the example mapping means this map binds to the *third* repetition.

L2100A_IL_DMG0501_RAC_ETH_CD	L2100A_IL_DMG0501_E02_RAC_ETH_CD	L2100A_IL_DMG0501_E03_RAC_ETH_C	L2100A_IL_DMG0503_CS_RC_ETH	L2100A_IL_DMG0503_E02_RAC_ETH_CD	L2100A_IL_DMG0503_E03_RAC_ETH_CD
9	A	8	B	7	C

DMG*D8*19890809*M**9>RET>A^8>RET>B^7>RET>C~

Example 6

L2300_DTP_STMNT_RD8_2

C:\serenedi\docs\specs\ 5010_837I_A2.html / Loop 2330:

L2300	DTP	Statement Dates		
03		L2300_DTP_STMNT_RD8_1	Start Date	Statement (RD8)
03		L2300_DTP_STMNT_RD8_2	End Date	Statement (RD8)

This is an example of a date range map. This map will always appear as a pair, and both maps together will encode a single element, in this way:

L2300_DTP_STMNT_RD8_1	L2300_DTP_STMNT_RD8_2
Jan. 1st, 2019	Jan. 30th, 2019

DTP*434*RD8*20190101-20190130~

Example 7

L2330C_02P3_REF_PVR_COMM_NR

The mapping documentation is found in the 837 P mapping guide:

C:\serenedi\docs\specs\5010_837P_A1.html / Loop 2330C:

L2330C - OTHER PAYER REFERRING PROVIDER (Inherited Loop Iteration & Value Qualified)
Mapping Prefix: L2330C_xx*DN* - Referring Provider
Mapping Prefix: L2330C_xx*P3* - Primary Care Provider

L2330C	NM1	Other Payer Referring Provider		
L2330C	REF	Other Payer Referring Provider Secondary Identification		
02		L2330C_xxyy_REF_STAT_LIC_NR	String	State License Number
02		L2330C_xxyy_REF_UPIN	String	Provider UPIN Number
02		L2330C_xxyy_REF_PVR_COMM_NR	String	Provider Commercial Number

This is an example of the Inherited Iteration and Value Loop Type. The four digits after the loop identifier (L2330C) indicate that this is the second loop iteration of the inherited parent loop, 2320, and that this map pertains to a Primary Care Provider (P3 qualifier listed in the mapping guide) iteration of the 2330C loop.

Example 8

```
STHDRX_PLB0301_PVR_ADJ_ID
```

The PLB segment mappings at the end of the 5010 835 transaction are unique in that they represent a cutout that's not in the normal data encoding path.

To present these mappings in a Flat interface, the STHDR and parent loops should be present in the data row along with the first PLB segment information, with NEWROW set to 1. For any subsequent iteration of the PLB segment for that transaction, NEWROW should be 0.

ENCODING VS. DECODING

Up to now, we have covered the essentials of how SERENEDI binds mappings from database tables and cells to defined elements within a supported EDI transaction. In this section, we will approach this problem at a higher level, and discuss how the business requirements of creating and parsing EDI transactions relate to the bidirectional SERENEDI translation engine.

When decoding EDI transactions, SERENEDI will create a mapping and assign values for every mapping it encounters. Note that "mapping" here is very different from "elements" because, as we see in the above examples, a single mapping can be half of an element, such as when encoding RD8 time spans, or it can encapsulate two elements in the EDI file, as is the case for every qualifier/identifier pairing.

Note that the CGIF2 Flat map will generally contain every single mapping present in the file, in every single row, starting at the ISA02 element at the outer envelope and going on to the GS loop, Transaction Set header loop, and so on into the deepest loop. Furthermore, a mandatory NEWROW column ends every Flat, forcing the engine to encode multiple cutout mappings instead of continuing to parse segments along the main data encoding branch of the hierarchy.

Along with the most common business mapping, many *scaffolding* elements are present as well – for example, the number of segments in the SE segment that ends a transaction will be decoded and parsed, and present in every single row of the transaction. These are generally ignored since these scaffolding elements do not directly represent business information. The problem here is that it conflicts with a very common scenario, which is to reprocess EDI files for a different trading partner or business purpose.

For example, say that an HMO has been collecting all of the Provider 837 Claim files for the entire year, but then the state dictates that these files must be resubmitted to the state's health department for analysis of certain health metrics on a populace scale. The state requires that the headers be changed and that certain data elements be altered or removed to accommodate its data requirements.

With SERENEDI, this may seem simple and straightforward – decode the original file to the Flat register, send that to a database table, UPDATE the columns to reflect the new header values, remove the columns for the maps the state does not want, then re-encode the file and send it to the state. But this approach will definitely fail.

The reason is that all the decoded scaffolding elements, like Number of Segments, are now being provided to the SERENEDI engine for encoding, and removing some data elements will alter the number of segments from the original file. The file

will be parsed by the state and rejected because the number of segments provided in the file does not match the number of segments actually present in the file.

The solution to this problem? Give SERENEDI *fewer* mappings so it can generate correct default values independently.

DEFAULTED SCAFFOLD ELEMENTS

SERENEDI can generate default mappings for the following **ISA** segments:

ISA01-ISA04 – If values are not provided for these maps, SERENEDI will default them to 00 and spaces. Note that no matter what, the first mapping provided to SERENEDI must contain the two-digit specification identifier.

ISA09 – The six-digit year time stamp will default to the current date.

ISA10 – The four-digit time stamp will default to the current time.

ISA14 – If no value is supplied, SERENEDI will default a value of P, meaning production.

IEA01 – This will be defaulted to the number of included GS/GE functional groups.

GS04 – This will default to the current eight-digit date stamp.

GS05 – This will default to the current four-digit time stamp.

GS08 – This will default to the correct specification identifier supplied in the initial two-digit specification mapping prefix.

GE01 – This will default to the number of Transaction Sets encoded.

ST03 – This will share the same value as GS08.

SE01 – This will default to the number of segments in the transaction.

BHT04 – This will default to the current eight-digit date stamp.

BHT05 – This will default to the current four-digit time stamp.

HL01-HL04 – The hierarchical level mappings will be automatically generated based on the situation of the data present within the transaction.

CAS02-CAS19 – SERENEDI will *shift* CAS mappings in groups of three to the left if there are data "bubbles" filled with unassigned values, such as when data is present for mapping CAS05/CAS06/CAS07, but no data is present for CAS02/CAS03/CAS04. This enables developers to assign specific business adjustment amounts to specific elements without worrying about creating a noncompliant segment because of missing elements earlier in the segment – SERENEDI will respect the contents of the data but not the specific position of the data in order to create a compliant segment.

Therefore, a "less is more" approach is necessary when recasting transactions for another purpose by removing all the elements defined above that pertain to scaffolding and letting SERENEDI choose the best values automatically.

FLAT INTERFACE

Encoding CGIF2 Flat to HKey

The HKey register is the internal SERENEDI representation of a HIPAA EDI transaction, but not yet generated into a text file. This section will dive into the process SERENEDI uses to translate a CGIF2 Flat register into the HKey register, which is very close to generating an actual EDI transaction.

When SERENEDI encounters a loaded CGIF2 Table, whether loaded from a database table, CSV file, or other source, it goes through the following process to translate this two-dimensional data source into a hierarchical data projection:

1. First, it obtains the specification from the first two digits of the mapping, the specification tag. These tags are defined in the *Appendix 1*
2. Second, all the mappings are sorted into hierarchical order, with the initial ISA mappings occurring first and the deepest hierarchy mappings put in the last place.
3. Third, data is scanned from left to right in sorted order – thus, the ISA mappings are scanned first in every row, then the GS mappings, and so on down the hierarchy.
4. Mappings are empty if they contain a database *null* value. String values are empty if they have a database null or zero-length string. Any loop that contains at least one non-empty mapping is considered as having data and will be encoded.
5. The first row encountered by the parser will have all non-empty loops encoded. For every subsequent row thereafter, every loop is compared to the corresponding loop in the previous data row. If a difference is detected, then that loop and *all* non-empty loops that are deeper in the hierarchy are considered "fresh" data.
6. For each database row, the NEWROW column must be present with a 1 or 0 value. If the value is 1, then the deepest standard data-carrying loop is considered fresh data and will not be compared to the previous database row. If the value is 0, then every non-empty cutout loop will be marked as fresh data and encoded into the HKey.

To give a simplified visual example of this process, consider the following table, which represents a simplified selection of loops in an 837 Institutional file:

ISA	GS	ST	2000 SBR	2300 CLM	2400 SVC	NEWROW
X	X	X	X	X	X	1
0	0	0	0	0	X	1
0	0	0	0	DELTA	X	1
X	X	DELTA	X	X	X	1

X – Data present and selected for encoding
O – Data present and *not* selected for encoding
DELTA – Data present that is distinct and different from the previous row

Each cell in this table represents a collection of individual mappings associated with each loop and presented to the SERENEDI parser. In this example, every loop is considered non-empty. Examining the top row, we see that *every* loop has data and every loop is selected for encoding. In the second row, every data element is exactly the same as in the previous row. The deepest non-empty standard loop is 2400 SVC, and it will automatically be selected for encoding since NEWROW contains 1 and therefore the deepest non-empty standard loop is automatically selected for encoding.

In Row 3 of the above example, the CLM loop contains at least one element that is different from the previous row. As a result, that loop and everything deeper are considered fresh data marked for encoding.

In Row 4, the Transaction Set header loop contains at least one element that is different from the previous row, and as a result, everything deeper along the hierarchy is marked as fresh data and encoded.

The order of the encoded loops in an EDI file, ignoring trailing envelope segments, will proceed like this:

ISA	Outer Envelope
GS	Group Header
ST	Transaction Set Header
2000 SBR	Subscriber Loop
2300 CLM	Claim Loop

2400 SVC	Service Line
2400 SVC	Service Line
2300 CLM	Claim Loop
2400 SVC	Service Line
ST	Transaction Set Header
2000 SBR	Subscriber Loop
2300 CLM	Claim Loop
2400 SVC	Service Line

Potential Pitfalls of CGIF2 Flats

For well-formed EDI transactions, this system allows SERENEDI to handle virtually any file, transform it into a representation that is straightforward for humans to work with using SQL database tools, and then transform it back into an EDI file. But what about *broken* EDI transactions?

In this case, what if an original set of claims and a *duplicate* set of claims data is sorted and then presented to the SERENEDI parser – how does SERENEDI handle this case? According to the rules given above, it's very predictable: since SERENEDI is expecting data to be completely sorted before seeing it, all duplicate claims will be aggregated together in sequential order and processed row-by-row by the parser.

Since there is no difference between the Original Claim and Duplicate Claim maps, it won't trigger a delta that will help the SERENEDI parser trigger the row as a new claim. However, since each database row is guaranteed to encode at least one loop, *all* the original and duplicate service lines for the claims will be encoded – leading to each claim having double the original number of service lines, and obvious imbalances in the Claim Charge amounts.

For more information on creating outbound EDI files, see the chapter "Creating Outbound EDI Files."

DECODING HKEY TO CGIF2 FLAT

The opposite of encoding an HKey (and by implication, an EDI transaction) is *decoding* an HKey to a Flat. Unlike the encoding step, this process is completely automated. The format of a HIPAA EDI transaction is rigidly predefined within the HIPAA implementation guides, and therefore every loop, segment, and element has a strictly assigned role. The way SERENEDI automatically creates a Flat register from an HKey register is pretty much the opposite of the encoding steps:

1. Descend into the hierarchical segments of the transaction and decode all mappings into a two-dimensional data table.
2. When the deepest loop encountered iterates or starts to ascend to a higher hierarchical level, trigger a new row and store all the encountered maps in the columns. Copy all parent loop maps. Mark the NEWROW column as 1.
3. When cutouts are encountered, they are stored and processed in line with the other maps *unless* they repeat more than once. If that occurs, create a brand-new row, copy the parent loop maps, and iterate only on the cutout segments, with each NEWROW field marked as 0.

At the end of the decoding process, a table that maintains proper sorting from highest hierarchy mappings to lowest hierarchy order is generated. And, assuming there were no integrity errors in the source EDI, immediately feeding this Flat table back into the encoder should yield a verbatim copy of the original EDI file.

HIERARCHICAL DATABASE INTERFACE

The Hierarchical Database (HDB) interface provides an alternative method to storing SQL query-accessible transaction data compared to the Flat interface. Instead of a single database table, one database table per loop is utilized in the transaction, joined together in parent-child relationships that exactly mirror the structure of the EDI transaction. To see the exact

relationship between parent and child loops in SERENEDI's somewhat customized implementation of the HIPAA implementation guide's hierarchies, see "Appendix: Specification Hierarchy Structures."

By default, the names of the HDB tables begin with HDB_5010, an underscore, the short specification name without any Addenda suffix, an underscore, and the loop short name. A collection of HDB tables for a specific transaction is called an *HDB tableset*.

For example: **HDB_5010_837P_L2320**

The layout of every HDB table begins with four columns:

PK_ID (int)	**Primary key**, auto-generated identity column
BIN_ID (int)	Foreign key reference to the BIN_LOG table
BIN_IX (int)	BIN Index, a numerically increasing index starting at 1 for every new BIN_ID
PAR_BIN_IX (int)	Parent BIN_IX, relates this loop's maps and data to the parent BIN_IX identifier

In the example above, this table could contain maps such as:

`L2320_01_SBR01_PYR_RESP_SEQ_NR`

The CGIF2 mappings are *very* similar to the Flat interface implementation, with one key difference: all loop iterations belonging to Single Iteration and ValueIteration maps are locked at 01. The loop iteration maps are redundant as the loop data structure itself encodes this information merely by having two Single Iteration loops parented to the same row. Besides this, the maps are functionally identical to CGIF2 Flat interface mappings.

The default table names can be overridden with a supplied prefix. In this case, the loop names will be added to the supplied prefix so that the loop data can be retrieved.

XML INTERFACE

The XML interface is ideal for making data consumable by "NoSQL" hierarchical database systems. The mapping rules for XML differ somewhat from the database-centric systems described above, with the loop and segment-element parts of the mapping split into two.

Similar to the hierarchical database system, loop iterations for qualified loops are not present in the XML Interface mapping system – these are implied naturally from the structure of the XML file. Thus, mappings that normally look like this:

L2330C_01DN_REF_PVR_COMM_NR	L2330C_01P3_REF_PVR_COMM_NR
123450004	123450005

. . . will be presented in XML like this:

```
<L2330C_DN>
      <REF_PVR_COMM_NR>123450004</REF_PVR_COMM_NR>
</L2330C_DN>
<L2330C_P3>
      <REF_PVR_COMM_NR>123450005</REF_PVR_COMM_NR>
</L2330C_P3>
```

For the purpose of brevity, we will assume all file paths are in Windows format. The base installation of SERENEDI contains the files shown at right.

serenedi

- **bin**
  - cnnstr.txt
  - license.key
  - SERENEDI2.dll
  - serenediStudio.dll
  - urls.txt
- **db**
  - serenediCore.sql
- **docs**
  - SERENEDI_OCI_Manual.pdf
  - LICENSE.txt
- **specs**
- **pipeline**
  - Pipeline.ps1
- **seed**

In the **bin** folder are the binaries required by the .NET Core environment; most of the files and subdirectories are various libraries used by .NET Core and SERENEDI. The **cnnstr.txt** file drives the connection to the serenediCore database; if it's not valid, or if no serenediCore database is installed there, then SERENEDI will not run. Setup for this file is explained in the installation instructions.

The **license.key** file is associated with your licensing tier and period. Replacing it can be done while the service is running, so if your licensing options change, it can be replaced in real time.

The **.dll** files are the prime .NET Core binaries that run the SERENEDI environment.

The **urls.txt** file drives the operation of SERENEDI Studio. This is explained in more detail in the installation instructions.

In the **db** folder, there is a SQL file that contains all the instructions needed to create the initial serenediCore distribution database on an Oracle server. It needs to be run just once per installation. There are two versions: One is utilized by the automated install script, and the other (serenediCore.sql) is used for manual schema creation.

The **docs** folder contains the manual, the licensing information, and various licensing files needed by the libraries SERENEDI uses.

The **specs** folder contains all the HTML files for the maps supported by SERENEDI.

The **pipeline** folder starts with a single SCORE script, Pipeline.ps1. When SerenediService is first run, it executes a single bootstrap event that calls on the pipeline system to initialize itself. If initialization is successful, a number of folders will be present relating to the built-in pipeline system in this directory.

The **seed** folder contains 14 seed files. These can be used to test different scenarios, and are created from the sample data and SQL Stored Procedures stored in the serenediCore distribution database.

SERENEDI TECHNICAL REFERENCE

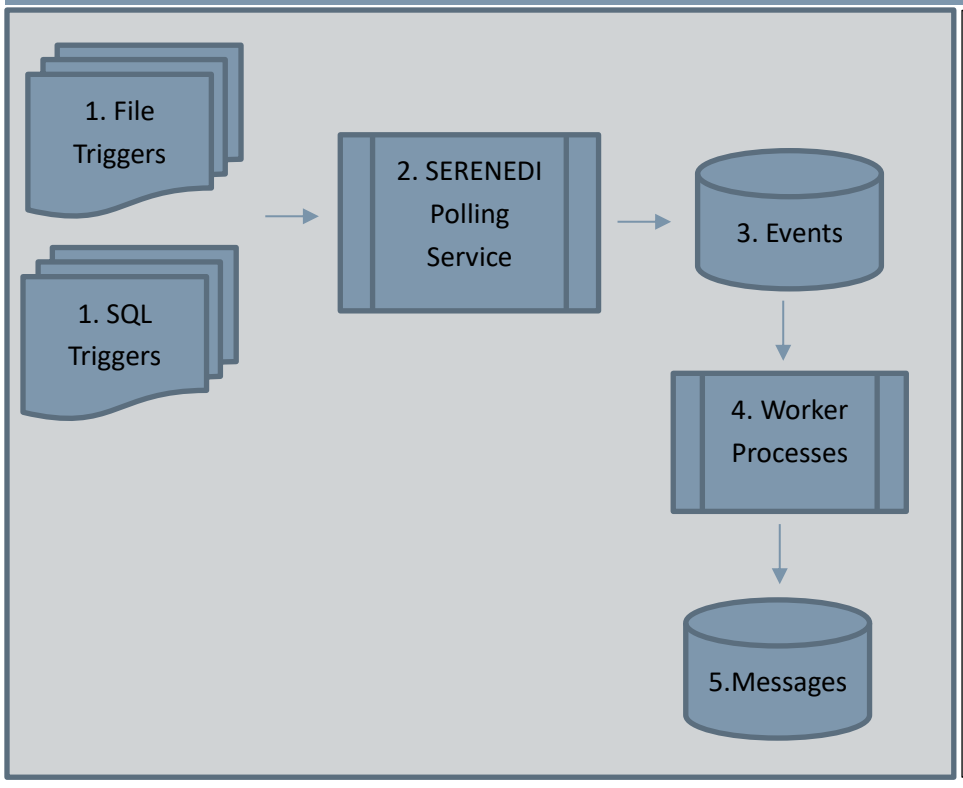
OVERVIEW

This technical reference is meant to give a full, detailed review of all technical aspects of SERENEDI. All aspects of the platform are covered so that you have a single reference point.

We start with a review of the CGIF V2 mapping system that is the core of all projections for HDB, CSV, Flat, and XML conversions. We go deep into the automation system and review the technology behind triggers, events, and workers. Then, we go into the architecture of the distribution database, serenediCore, and review each table and column.

Next, in the SCORE Script reference, we review the Serenedi API commands that comprise the SCORE scripting system, along with error messages. Afterward, we review the REP Code user-extensible Rules Engine codes. Finally, we review the hierarchical relationships of the loops within each specification – which is critical for understanding how HDB BIN data is committed to the database.

Event System



Event Life Ccycle

1. Triggering conditions are set up in the BIZ_TRIGGER table.

2. The SERENEDI Polling service continuously scans for firing conditions.

3. When a firing condition is met, a new entry is made in the BIZ_EVENT table.

4. A background Worker Process is assigned to the new event, and processes it.

5. The event result is stored in the BIZ_EVENT table and messages are stored in the BIZ_MSG table.

!

SERENEDI uses PowerShell Core with custom Cmdlet extensions and runtime environment. In the rest of the manual, these are referred to as **SCORE scripts**.

The SERENEDI Event system is designed to make automation of healthcare integration processes simple and straightforward. It begins with the triggers set up in the BIZ_TRIGGER table. This table defines the criteria by which events are generated, and also specifies the PowerShell Core SERENEDI ("SCORE") script to execute when the trigger is fired. A background process constantly scans the firing criteria, and if the criteria are met, then a new event is generated. New events can be generated by external database events as well.

In the background are a number of worker processes that scan the BIZ_EVENTS table for new work to do. When a worker process manages to take exclusive ownership of an event, it runs the SCORE script associated with that trigger and sets certain global variables according to the firing criteria and type of event. It's important to note that SCORE scripts generally run in *parallel* with one another unless a trigger is set up for serial execution.

SCORE SCRIPT SYSTEM

SERENEDI contains an embedded scripting system called PowerShell Core. It extends the default library with a number of *cmdlets*, which is PowerShell Core terminology for custom functions. Each time a SCORE script runs, a new SERENEDI Session state object is spawned that encapsulates the main registers used for EDI translation and conversion. It also contains the "MsgLog," register, a running list of messages generated throughout execution of the script; at the end of execution, these messages will be added to the BIZ_MSG table for later reference.

Each event has up to four string arguments that are available to be used within the SCORE script as in-session variables. The maximum runtime of a SCORE script is one hour. If that threshold is exceeded, the background service will assume it is a hung process and stop it. This prevents rogue events from tying up individual worker resources dedicated to processing the events.

SCORE scripts are aimed at a single business process, with one SCORE script servicing one business process. SCORE scripts are designed to be run in small, independent sections. And, because the SERENEDI scripting environment enables the creation of triggers and supporting objects, a single SCORE script is capable of bootstrapping all the objects needed to run a full multi-trigger business process from scratch: a single initialization handler creates the triggers and directories, and then different sections independently handle the different triggers.

This way, the SCORE script can easily be deployed and redeployed to development, tested thoroughly in QA, and finally deployed to production. Furthermore, keeping the execution of the separate sections short makes debugging much easier.

TRIGGERS

A SERENEDI trigger is broadly defined as the initial condition established to begin work on a specific item. Because the events fired by the trigger are meant to run in parallel, all the work it accomplishes should be isolated from interfering with other triggers, or from being influenced by other events fired by the same trigger. The SERENEDI system architecture is specifically designed to let dozens of concurrent processes work without interference.

There are four ways to trigger an event:

1. **Direct Injection** – Bypasses the normal trigger system to directly execute a SCORE script with parameters. Direct Injection can be accomplished with either SQL inserts or XML messages.
2. **SQL Trigger** – Sets specific criteria based on SQL results to fire an event.
3. **Upload Trigger –** Sets specific criteria based on moving files from one folder to another to fire an event
4. **File Trigger** – Sets specific criteria based on the presence of files in a specific folder to fire an event.

Direct Injection

Direct Injection of events means you are creating an event that is not tied to a specific trigger. Since the trigger contains information about the SCORE script to execute, then the path and filename of the SCORE script must be included in the fourth argument parameter for this event, which leaves the first three arguments available to pass information to the script.

One example is to present the way the whole pipeline environment is able to bootstrap during the initial install. Before the bootstrap, no triggers are defined at all in the database environment:

```
INSERT INTO BIZ_EVENT (EVENT_DATA3, EVENT_DATA4) SELECT 'INITIALIZE',
'C:\SERENEDI\PIPELINE\PIPELINE.PS1'
```

Direct Injection events can be used anytime you don't need a complex triggering architecture, and only need to run business processes on an as-needed basis. When the event is finished executing, you may review the logs in BIZ_MSG that are tied to the new BIZ_EVENT row.

XML Injection

The XML Injection path enables users to create events and see the results using only XML files. The XML file must have a unique name, and it must contain a single XML message formatted similar to the following:

```
<CtrlMsg>
        <trig>1</trig>
        <crit>FIRING_CRITERIA</crit>
        <arg1>Argument1</arg1>
        <arg2>Argument2</arg2>
        <arg3>Argument3</arg3>
        <arg4>Argument4</arg4>
</CtrlMsg>
```

The 010 Event pipeline picks up the message and pushes it into the BIZ_EVENTS table, where a worker process picks it up. Then, when it has finished execution, the resulting messages are pushed into the BIZ_MSG message log, to an XML file in the output folder.

If the **trig** parameter is not supplied, then the **arg4** parameter is used as the path and filename to a SCORE script that runs. The first three arguments are passed to the SCORE script when it starts execution.

SQL Trigger

This fires a trigger when an SQL statement returns a result of 1. The actual SQL itself is supplied in the FIRE_LOGIC column of the trigger and must be prefixed with **SQL:**. If this condition becomes true during polling, the event is fired. If any of the FORCE_ARG1 – FORCE_ARG3 parameters are set within the trigger, they will be set within the resulting event. If FORCE_ARG4 is set, it will be parsed as a Bin Endpoint Alias and the query will be executed against that database endpoint. This allows external database servers, such as Oracle servers, to trigger new events.

File Trigger

The file trigger system in SERENEDI is quite flexible. First, there are two primary modes of operation, ARCHIVE and UPLOAD mode. UPLOAD mode requires two folders, the Initial folder and the Source folder. The Initial folder is where files are initially dropped; when the background SerenediService is able to move a file from the Initial folder to the Source folder,

then the event on that file is fired. The only dependency is that the file does not exist in the Source folder prior to the event.

ARCHIVE mode will poll the BIZ_EVENT system and the file system every time the trigger is polled. Any new files that have *not* been linked to an event will be fired as new events.

Fire Logic

Fire Logic filters a triggering condition of Upload, Archive, and SecureFTP Archive file triggers in one of four ways. The filters can be *stacked* with the pipe character, allowing a rich set of criteria.

FILTER

The filter is a flexible tool to tightly control the types of files that fire the trigger. For example:

FILTER:*.835

The above filter will trigger on files *only* if they end in the .835 extension – all other files will be ignored.

FILTER:ELIG??????.834

The above uses the ? wildcard character to establish that the filtered file must begin with ELIG, end with .834, and have six indeterminate characters in the middle to fire the trigger.

In some cases, it's necessary to process *file sets* instead of individual files – that is, a file should be processed if and only if it's accompanied by a set of files with a common naming protocol. SERENEDI can accomplish this by stacking the FILTER directive with different criteria. This behavior is ideal for ARCHIVE triggers since files don't need to be moved to fire the trigger, and thus the set is kept together.

Example:

FILTER:Z1*.TXT|FILTER:Z2*.TXT

If two files named Z1XYZ.TXT and Z2XYZ.TXT arrived, then this filter would allow the Z1XYZ.TXT to fire the trigger. If the second file was named Z2XYZZ.TXT, the file would *not* trigger because the wildcard pattern is not common to both files.

TIMESTAMP

Technically, this is not a filter so much as it is a modification to an Upload trigger. In this case, the timestamp value is treated as a C# DateTime Format string that will be added to the end of a filename (and prior to the extension) when the file is moved to the Source folder. This enables Upload triggers to operate on new files with old filenames – by suffixing a unique date/time stamp, the old filename becomes a new one, and the file can fire the trigger.

Example:

TIMESTAMP:yyyyMMddHHmm

If an Upload trigger filename started as CLINIC_001.835 and was uploaded on December 21, 2019, 12:51 a.m., it would end up as CLINIC_001201912210051.835, as an example.

SQL

This sets the SQL that will be used in an SQL trigger. For SQL triggers, the firing condition is when this SQL executes on the SERENEDI database and results in an integer 1 value.

Example:

SQL:SELECT 1

This will result in an SQL trigger that executes every time it is polled. Note that only SQL triggers can execute triggers external to the serenediCore database; the FORCE_ARG4 field of the trigger is used for the BIN Endpoint Alias.

SCORE SCRIPTS

SCORE scripts – aka SERENEDI PowerShell Core scripts – leverage the cross-platform basic functionality of PowerShell Core to expose virtually every part of the SERENEDI engine to the developer. If your business needs go beyond the basic transformations supported by the built-in pipeline system, then you'll need to learn about developing SCORE scripts.

The ideal way to develop SCORE scripts is with the Visual Code environment, as that allows a rich debugging and development system. If you want to test some simple SCORE commands *without* installing this environment, there is a simple REPL (read-eval-print loop) command line interface that ships with SERENEDI. To execute it, go to the BIN directory and execute:

```
dotnet SERENEDI2.dll REPL
```

This will initialize the environment and allow you to test the effects of various commands using a simple command-line interface.

To get the most out of the SCORE system, it's important to keep these points in mind:

PORTABLE

Portable scripts are ones that can easily be deployed from one server to another, especially from a development environment to a QA or production environment. This means:

- Absolute paths should be avoided. The SERENEDI installation path may not be the same on the server you are targeting. The $ wildcard used within SERENEDI directory names will always resolve to the SERENEDI/pipeline directory as it exists in that particular environment. Because this wildcard is unknown to normal PowerShell Core commands, you could set the $ pipeline variable in this example, and use that instead:

```
$pipeline = (Join-Path $basePath 'pipeline')
```

- You'll need to be careful with path name separators. For example, Unix pathnames follow a *forward-slash*, while Windows uses a *backslash* character to specify paths. By using the Join-Path command as shown above, you can indicate a full path that will work regardless of whether the target server is on a Unix or Windows platform.
- Use SCORE Environment commands for setup. Because triggers, endpoints, and SecureFTP sessions can all be initialized using only SCORE commands, this also means that a single INITIALIZE subroutine in your workflow can fully set up and prepare the SERENEDI environment from scratch. You could also have a DEINITIALIZE subroutine for takedowns, but be aware that once messages and events are tied to your new triggers, it's not a simple matter of deleting everything – you'll need to delete the messages, then events, before the relational constraints allow you to delete the triggers.

PARALLEL

- SCORE scripts are meant to be executed in parallel – many processes executing the same script, but with different arguments. The background BIN system is engineered to feed data from dozens of processes simultaneously without database locks or choke points. This means that *generally*, SCORE scripts should process at most one file at a time. Although PowerShell Core has looping mechanisms, you will get much better performance from the environment if you write short scripts that operate on one file at a time, then exit. This way, the SERENEDI load-balancing system can make the most of system resources.
- This also means you should avoid having SCORE scripts execute long-running database operations. SERENEDI has safeguards to prevent SCORE scripts from running too long, and will kill the process if it exceeds the built-in four-hour time limit.
- SERENEDI will have a set number of worker processes running in the background, waiting for new work to appear in serenediCore. This is usually a quarter of the licensed workers or a quarter of the available CPU cores, whichever is less. Once it starts to become busy, SERENEDI automatically launches the full available number of worker processes (the number of available CPU cores *or* the licensed number of cores, whichever is less) until there is no more work to do, at which point it returns to "idle" mode.

PIPELINE WALKTHROUGH

In this section, we will go into much more detail about one SCORE script that is provided with the distribution: Pipeline.ps1. The main reason is that this single SCORE script does the following things:

1. It creates the entire pipeline folder hierarchy as well as all pipeline triggers
2. It partitions functionality to serve all the pipeline triggers in individual sections

This is a good model to follow when creating custom SCORE scripts for your own applications. With a dedicated installation function, the SCORE script becomes *portable*. This same script could be tested and deployed on a development server, then deployed on a QA server, then on a production server, all with the same steps.

By handling multiple triggers, it's possible to maintain a single script to handle multiple, related business processes and have a single point of maintenance.

Setting the Base Directory

…

```
$base = (Join-Path $basePath 'pipeline')
```

…

The $basePath is a predefined SCORE variable that represents the SERENEDI installation directory. Many people would choose to simply add the pipeline folder to the base directory, like so:

$base = $basePath + '\pipeline'

Since UNIX systems and Windows systems use different characters for path separators, this is *not* cross-platform. The Join-Path function, however, will work reliably to join two paths regardless of platform.

Installing the Environment

…

```
if ($eventData3 -eq 'INITIALIZE')
{
```

```
    $pipebase = (Join-Path $base '001_Normalize')
    mkdir $pipebase
    mkdir (Join-Path $pipebase '01_in_edi')
    mkdir (Join-Path $pipebase '02_done_edi')
    mkdir (Join-Path $pipebase '03_out_edi')
    mkdir (Join-Path $pipebase '04_err_edi')
    sapi-EnvTriggerUpsert -TriggerName 'PIPE001_NORMALIZE' -Script '$\Pipeline.ps1' -TriggerType
'LOCAL_UPLOAD' -InitFolder '$\001_NORMALIZE\01_in_edi' -SourceFolder '$\001_NORMALIZE\02_done_edi' -
PollInterval 30 -IsEnabled $True -ForceArg3 'PIPE001_NORMALIZE'
```

…

First, the *third* argument is checked for the Initialize command. This is important because when the serenediCore database is first set up, prior to SerenediService running, there are no triggers at all – only a single BIZ_EVENT set up to execute this script in *immediate mode.* Once SerenediService first starts to run and launches a worker process, this initialization event will be the first thing to be processed.

This small section sets up the environment for the *Normalize* pipeline. First, it makes the directory hierarchy for the pipeline, and then it creates the trigger. Note the use of the $ in the command – this is shorthand for C:\serenedi\pipeline or wherever SERENEDI is installed to, and is cross-platform compatible. The **sapi-EnvTriggerUpsert** command will either update an existing trigger with the same name or, if the trigger does not exist, create it from scratch. The script is set the same for all of these triggers: **$\Pipeline.ps1**. The LOCAL_UPLOAD trigger type is a reliable way of processing incoming EDI files, as the trigger cannot fire unless the file has been successfully moved from the Init to the Source folders. The Poll Interval is set to 30 seconds, the event is enabled, and, finally, all events created with this trigger will have Argument 3 set to PIPE001_NORMALIZE. This is the how the SCORE script will differentiate this trigger from all of the other triggers executing this same script – via Argument 3.

Handling the Event

…

```
# NORMALIZE Pipeline
# 01_in_edi - EDI file to ingest
# 03_out_edi - Reprocessed and Normalized EDI file
# 04_err_edi - Errored EDI file

if ($eventData3 -eq 'PIPE001_NORMALIZE')
{
    sapi-SegPoolFromFile -Filename $eventData1
    sapi-SegPoolToHKey
    sapi-SegPoolFromHKey

    if ((sapi-FetchVar -Value "CRIT_ERR") -eq $false)
    {
        $newPath = (Split-Path (Split-Path $eventData1 -Parent) -Parent) + '\03_out_edi\' +
[System.IO.Path]::GetFileName($eventData1)
        sapi-SegPoolToFile -Filename $newPath -Formatting '*~>^' -bolCR $true -bolLF $true
    }
    else
    {
        $newPath = (Split-Path (Split-Path $eventData1 -Parent) -Parent) + '\04_err_edi\' +
[System.IO.Path]::GetFileName($eventData1)
        Move-Item -Path $eventData1 -Destination $newPath
    }
}
```

This small set of code is the entirety of the Normalize pipeline handler. After checking Argument 3, first the file is loaded from the **$eventData1** variable. This variable reflects EVENT_DATA1 from the BIZ_EVENT table, as set by the LOCAL_UPLOAD trigger type, when it is able to successfully move a file from the Init folder to the Source folder as defined in the trigger.

From there, it translates *to* the HKey register and then *back* from the HKey register. This will set all the encoding options – such as the element and segment separators – to the same value, and ensure all files are formatted the same way with respect to carriage returns at the end of segments, and so on. Furthermore, any minor deviances from the HIPAA Implementation Guide specifications will get filtered out in this step.

Assuming that the file did not have any critical errors, then **sapi-FetchVar -Value "CRIT_ERR"** will return a False Boolean value, and the **sapi-SegPoolToFile** command will re-create the file in the 03_out_edi folder.

If there *was* a critical error, then the original file will be moved from the 02_done_edi folder to the 04_err_edi folder.

Creating Outbound Transactions

The SEED system within SERENEDI is designed to fulfill the following design objectives:

1. Provide a set of sample HIPAA-compliant EDI files that can be used for testing and experimentation and do *not* contain Protected Health Information
2. Provide a set of fictitious managed care data that can be used to generate these seed files
3. Provide the SQL objects that can be used to project this data to SERENEDI so it can generate these EDI files

Every developer knows about the "tyranny of the blank page" – the need to create complex processes from scratch, starting from nothing. The SEED system built into SERENEDI ensures that, for whatever outbound specification you are trying to create, you have a starting point. Since HIPAA EDI files are necessarily concerned with transmitting managed care data, we needed to create a rich set of sample tables that mimic a simple managed care system so these seed files have information that makes them look and feel like normal EDI files. The clue that these tables do not actually contain any PHI is in the last names: they are names of ethnically diverse foods.

This sample data is consumed by the 14 stored procedures in the distribution database, which are described in more detail at the end of this chapter.

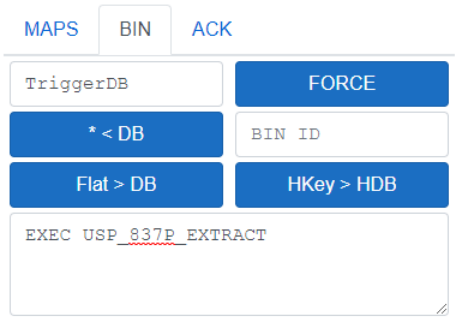
Each stored procedure translates the sample data and projects it to a CGIF2-compliant data extract that SERENEDI can turn into an EDI file. Therefore, if you'd like to quickly create a new EDI extract process, you can create a copy of an existing stored procedure and alter the extract so that instead of pulling from the sample tables, it pulls data from your managed care system sources. If the fields you need are not present within the extract, you can look up the HIPAA EDI equivalent in SERENEDI in the html files located under the /serenedi/docs directory.

Note that these examples are all data-ready to project data as a Flat register, but this is not your only option. Some situations – especially those that involve highly repeating cutout loops – are more difficult to express in the Flat projection than with the Hierarchical DB projection. You could artificially create a new HDB BIN entry in the BIN_LOG table, programmatically make insertions into the HDB tables using the BIN_ID to tie the data together, and then use the 008_BINToEDI pipeline to export the file to the file system, just as if you had imported the BIN normally. However, using the Flat export mechanism is easy and straightforward in most situations, and with stored procedures, SERENEDI can go directly from procedure to an EDI file in the file system.

The fastest way to generate an EDI file using these sample EDI extracts is simply to create a new BIZ_EVENT row that requests the 008_BINToEDI pipeline to generate a file from a dynamic data source. This generates a SEED_837P.TXT file and places it in the SERENEDI base directory:

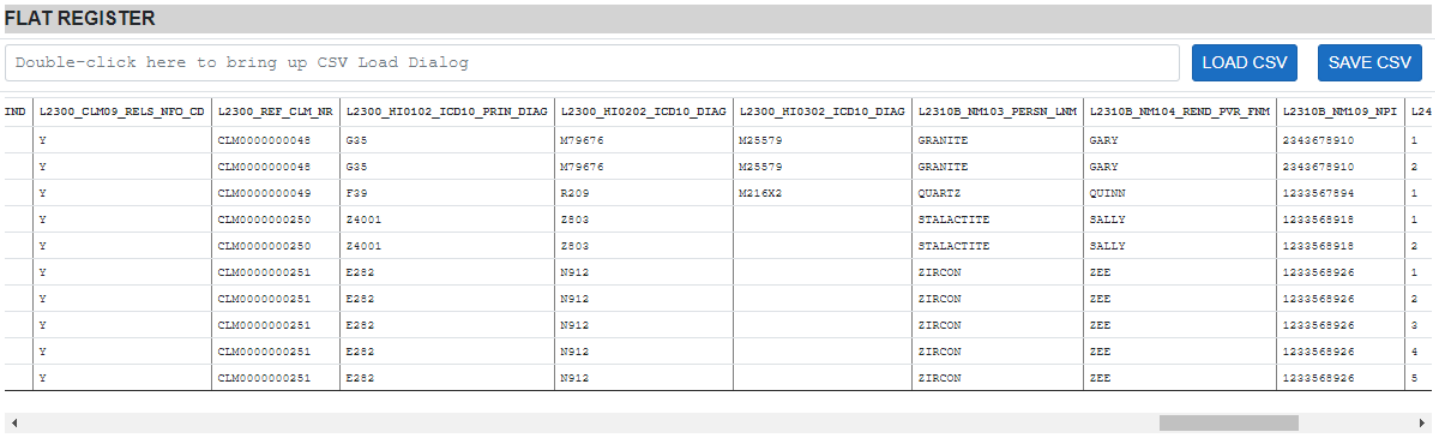
```
INSERT INTO BIZ_EVENT(BIZ_TRIGGER_ID, EVENT_DATA1, EVENT_DATA2, EVENT_DATA3) SELECT
BIZ_TRIGGER_ID,'EXEC USP_837P_EXTRACT','c:\SERENEDI\SEED_837P.TXT', 'PIPE008_BINToEDI'
FROM BIZ_TRIGGER WHERE TRIGGER_NAME='PIPE008_BINToEDI'
```

Another way to do this is by using SERENEDI Studio. This gives you several advantages during development: you can load the Flat register and then immediately decode it, thus validating it against the many business validation rules built into the SERENEDI environment. From SERENEDI Studio:



First, click on the BIN tab, enter EXEC USP_837P_EXTRACT in the bottom text box, and then click the blue * < DB button above the text box.

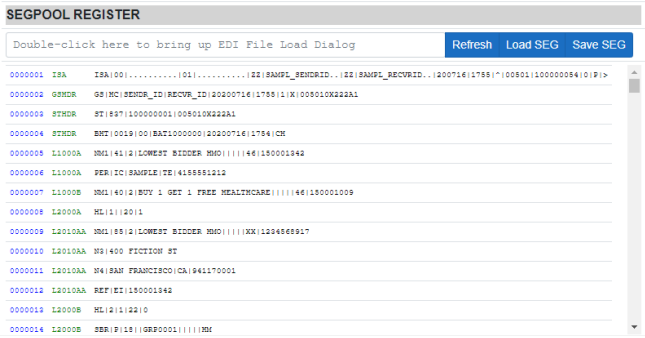
Data then is loaded into the Flat register, like so:



IND	L2300_CLM09_RELS_NFO_CD	L2300_REF_CLM_NR	L2300_HI0102_ICD10_PRIN_DIAG	L2300_HI0202_ICD10_DIAG	L2300_HI0302_ICD10_DIAG	L2310B_NM103_PERSN_LNM	L2310B_NM104_REND_PVR_FNM	L2310B_NM109_NPI	L24
	Y	CLM0000000048	G35	M79676	M25579	GRANITE	GARY	2343678910	1
	Y	CLM0000000048	G35	M79676	M25579	GRANITE	GARY	2343678910	2
	Y	CLM0000000049	F39	R209	M216X2	QUARTZ	QUINN	1233567894	1
	Y	CLM0000000250	Z4001	Z803		STALACTITE	SALLY	1233568918	1
	Y	CLM0000000250	Z4001	Z803		STALACTITE	SALLY	1233568918	2
	Y	CLM0000000251	E282	N912		ZIRCON	ZEE	1233568926	1
	Y	CLM0000000251	E282	N912		ZIRCON	ZEE	1233568926	2
	Y	CLM0000000251	E282	N912		ZIRCON	ZEE	1233568926	3
	Y	CLM0000000251	E282	N912		ZIRCON	ZEE	1233568926	4
	Y	CLM0000000251	E282	N912		ZIRCON	ZEE	1233568926	5

If the above is successful for your custom extracts, then this means all of your fields are CGIF2 compliant, and it is *likely* that SERENEDI can create an EDI file from this. To carry the process further, you will need to press some more buttons:

First, click HKEY < FLAT to load the HKey register from the Flat register that you just loaded. Then, click SEG < HKEY. You should see the EDI segments in the lower left corner. However, this hasn't subjected the file to the extensive validation checks. To do this, click SEG > HKEY. The process of transforming the SegPool register *back* to the HKey register also triggers all the validation checks built into the system.



The green loop names embedded in the file indicate that the file was decoded, as this information is only tied to the segments when it is successfully decoded to the HKey register. If the status window shows a count of 0 messages, then SERENEDI believes it is a compliant file:

```
SEGPOOL     0005354 segs
HKEY        Loaded : 5010_837P_A1
FLAT        085 x 0000889
Messages    000000 msgs
```

If there are messages, they should show up in red highlights next to the segment or loop that triggered the issue. If you have trouble viewing the messages because the resulting SegPool is too big to display, you can always use the RunBox command window to send the current message log to an HTML file in the file system:

```
sapi-MsgLogToFile -Filename 'C:\serenedi\msgs.html'
```

Tips for Creating Outbound EDI Files

1. There are two unique paths to creating new outbound files. One way is to insert a new record into the BIN_LOG table, record the new BIN_ID, and use that to populate new records in pre-established FLAT or Hierarchical BIN tables. Whatever SQL you use will depend on the data you are pushing to the BIN tables. The second way is to create a stored procedure that emits that data you need.

2. Usually, the best approach to creating new outbound specifications is to create them one field at a time, from the outer envelopes down to the deepest loops, and test the encodings step by step. Since encoding a file just takes a second, you can use the above steps to repeatedly test the maps and extract logic to make sure the segments are generated in the way you expect.

3. Be aware that each field in SERENEDI is associated with four cardinal data types: integer, decimal, string, and date/time. The types are provided for each mapping in the HTML files under the docs directory so you know what data type each column is expecting.

4. SERENEDI will not encode a loop unless at *least* one non-null data value is present in that loop. If you are getting invalid loops in the outbound file, check the data extract to make sure you aren't accidentally sending any non-null data elements for that loop. If you examine the sample extracts, you'll see several cases where entire sets of fields are encased in CASE WHEN <logic> THEN <value> ELSE NULL END. This is a way to ensure that the loop will only occur for a specific condition, and at no other time.

5. Remember that this encoding/decoding process is completely *two-way* – so if you're having trouble conceptualizing the data maps you need to create a certain set of segments, it might be easier to copy one of the sample files, manually edit it to add the segments you need, and observe how it decodes to the CGIF2 Flat space. This will guide you toward the best way to project your business data to achieve that result.

Common Attributes of the Seed Extracts

There are 14 stored procedures within the distribution database that define the seed extracts. At the very beginning, the distribution database increments the Interchange Control Number record in the SAMPL_HEADER that is linked to that transaction. This ensures that every file generated has a unique ICN. Then, it projects the sample data to create a compliant EDI transaction for that specification. A critical ORDER BY statement comes at the end. Although the extract itself could be placed in an SQL View, SQL Server does not honor ORDER BY clauses within Views, and therefore stored procedures are the most reliable way to ensure the data is in the proper order for encoding.

Also, each specification remaps the first field, ISA_ISA02_NO_AUTH_NFO from the SAMPL_HEADER table, to a different name that incorporates the specification tag. Without this specification tag, SERENEDI will not know what transaction these maps belong to, so this is quite crucial.

In each of the remaining sections in this chapter, we will discuss a different extract. To see the source code of the extract itself, you can right-click on the stored procedure and select Script Stored Procedure As, then Create To from within SQL Server Management Studio.

USP_270_EXTRACT

This stored procedure projects a group of subscribers and dependents in a mock 270 eligibility request file. It joins the Payer Provider ID provided in the Sample Member data to the Sample Provider table, and the ORDER BY clause at the end ensures data is sorted by these providers first, and then by the Member ID number. It generates 120 records.

USP_271_EXTRACT

The 271 extract is almost exactly the same as the 271 extract, except that it adds a Plan Begin date of 2019-01-01 for all of the 120 records it returns.

USP_276_EXTRACT

This query hits the sample tables to generate a Health Care Claim Status Request transaction for 889 claim lines. Some things to note about this extract:

- Because there are members without dependents defined in the sample tables, the 2200D and 2210D loops should only be present when the member is also the subscriber; the 2200E and 2210E maps should be completely nulled out to prevent an erroneous loop being encoded. This is accomplished with the CASE WHEN DOB1 IS NOT NULL THEN … ELSE NULL END logic for the 2100E/2210E maps – it ensures the claim information is only transmitted when DOB1 (Dependent Date of Birth in the subquery) has a non-null value, meaning that member is a dependent.
- The subquery is a way to bifurcate the member population into subscribers and dependents so that the appropriate logic branches can be used for each claim line.

USP_277_EXTRACT

This extract is a mirror of the 276 extract, simulating a response to a claim inquiry. Since the dependent date of birth is not sent back, the dependent ID card is checked for a non-null value to verify whether to send the 2200E loops.

USP_277CA_EXTRACT

This is a Claims Acknowledgment extract. Instead of sending, unlike other files, it requires that the total submitted charges for claims be submitted at a header level and member level. This is the responsibility of the two subqueries that generate the BLLR_CLM_CHG_AMT and the CLM_CHG_AMT fields: generate a sum total of all claims by biller and a sum total of claims by member.

USP_278_REQ_EXTRACT

This is a Health Care Services Review – Request for Review transaction. For this specification, each member to be reviewed is sent as a different transaction with its own ST/SE envelope. The Place of Service code filter restricts the query results to Inpatient claims. The subquery gathers Place of Service and minimum service dates for this extract that go in the 2000E loop.

USP_278_RESP_EXTRACT

This query provides a member-level response to the Health Care Services Review, so it does not need to encode any claim lines and gives a simple, canned response.

USP_820_EXTRACT

This is a Payroll Deducted and Other Group Premium Payment for Insurance Products extract where the payer is being billed $10 for every member (as shown in the L2300B_RMR04_DTL_PRM_PMT_AMT mapping). This file is split into six transactions, one for every payer, and every member is assigned a unique invoice within the file through the use of a ROW_NUMBER() command.

The subquery counts the members per payer so it can generate a header-level invoice amount for the whole bill.

USP_820X_EXTRACT

This Health Insurance Exchange Related Payments file is similar to the 820 extract above, except it needs to transmit policy dates. All policy dates are defaulted to 2020-01-01 to 2020-12-31.

USP_824_EXTRACT

This Application Reporting for Insurance specification reports a single Transaction Accepted status for a sample transaction.

USP_834_EXTRACT

This is a sample eligibility extract that uses hard-coded benefit begin and end dates. It transmits subscribers first, followed by dependents later in the file, and will only transmit the member address if the member is also a subscriber.

USP_835_EXTRACT

This is a sample 835 extract that transmits payment information for all the claim lines present in the sample data. The extract-stored procedure is a bit complex because of the following factors:

- The Transaction Header must contain a valid sum amount for all the claims within the transaction
- The claims must also transmit accurate total payment information
- The presence of the various Patient Responsibility amounts affects the presence and composition of CAS claim adjustment segments

The first subquery that generates CHK_PMT_AMT generates the transaction-level payment amount, whereas the second subquery generates the sums for Claim Charge Amount, Claim Payment Amount, and Claim Patient Responsibility Amount, as well as the minimum and maximum service dates per claim.

The real complexity lies in the UNION subquery within the CAS_DTL inner join. SERENEDI treats CAS segments as *cutouts*, which means repetitions of these segments are encoded vertically as different rows instead of as new columns. To keep the cutout information related to the claim detail line in the previous row, the NEWROW column is set to 0. This flag basically means “The only relevant information in this entire database row is the cutout mappings.” This NEWROW column is used only for flat-formatted extracts.

This means that our 835 extract must keep track of whether it is the *first* CAS segment (NEWROW = 1) or *additional* CAS segments (NEWROW = 0). The CAS_DTL clause establishes this logic: If there is a Claim Line Patient Responsibility amount above 0 (copay + coinsurance + deductible), then it will emit a CAS*PR segment, fill it with the appropriate information, and set the NEWROW to 1, and then any remaining difference will be transmitted as a successive CAS*CO segment where NEWROW is set to 0. If there is no Patient Responsibility amount, then only the CAS*CO segment is transmitted with a NEWROW set to 1 (it’s the only Claim Adjustment segment).

The CAS*CO will vary depending on whether there is a *claim withholding* flagged for this claim line. If so, it will be encoded with a 104 Remittance Adjustment Reason Code, and the remaining balance will be encoded in the same segment with a reason code of 45.

Note that within the CAS*PR encodings, the deductible, coinsurance, and copays are all assigned discrete slots within the CAS segment – CAS03, CAS 06, and CAS09 adjustment amounts. For claims with only a copay amount, this could lead to an invalid CAS segment as it would leave the CAS03 and CAS06 elements empty. As a convenience, SERENEDI will automatically “fill” in these earlier elements if they are left blank and higher elements are filled in – this check is done only on CAS segments.

At the end of the extract, all lines are sorted by Provider ID, Claim ID, Claim Detail ID, and NEWROW in *descending* order. This way, NEWROW = 1 will always occur before NEWROW = 0 rows for additional CAS segments, which is correct.

USP_837I_EXTRACT

USP_837P EXTRACT

The 837 Institutional and Professional extracts share enough in common that they will be discussed here together. Three different situations are demonstrated by this extract:

1. Subscribers with claims
2. Dependents with claims
3. Both subscribers and dependents with claims

In the first case, these extracts simply omit the 2000C patient loop for subscribers, and the claim and claim lines are encoded in a straightforward fashion.

In the second case, the extract gives the subscriber information, followed by the 2000C patient loop that describes the dependent information, followed by the claim.

In the third case, the subscriber information is first transmitted along with associated claims. Then, the subscriber information is relayed again, and each dependent is transmitted in separate iterations of the 2000C patient loop, followed by claims for that dependent. The initial HL**22 loop for the subscriber appears only once for all the dependents. This sequence is described in detail starting on p. 30 of the 837 Institutional Implementation Guide.

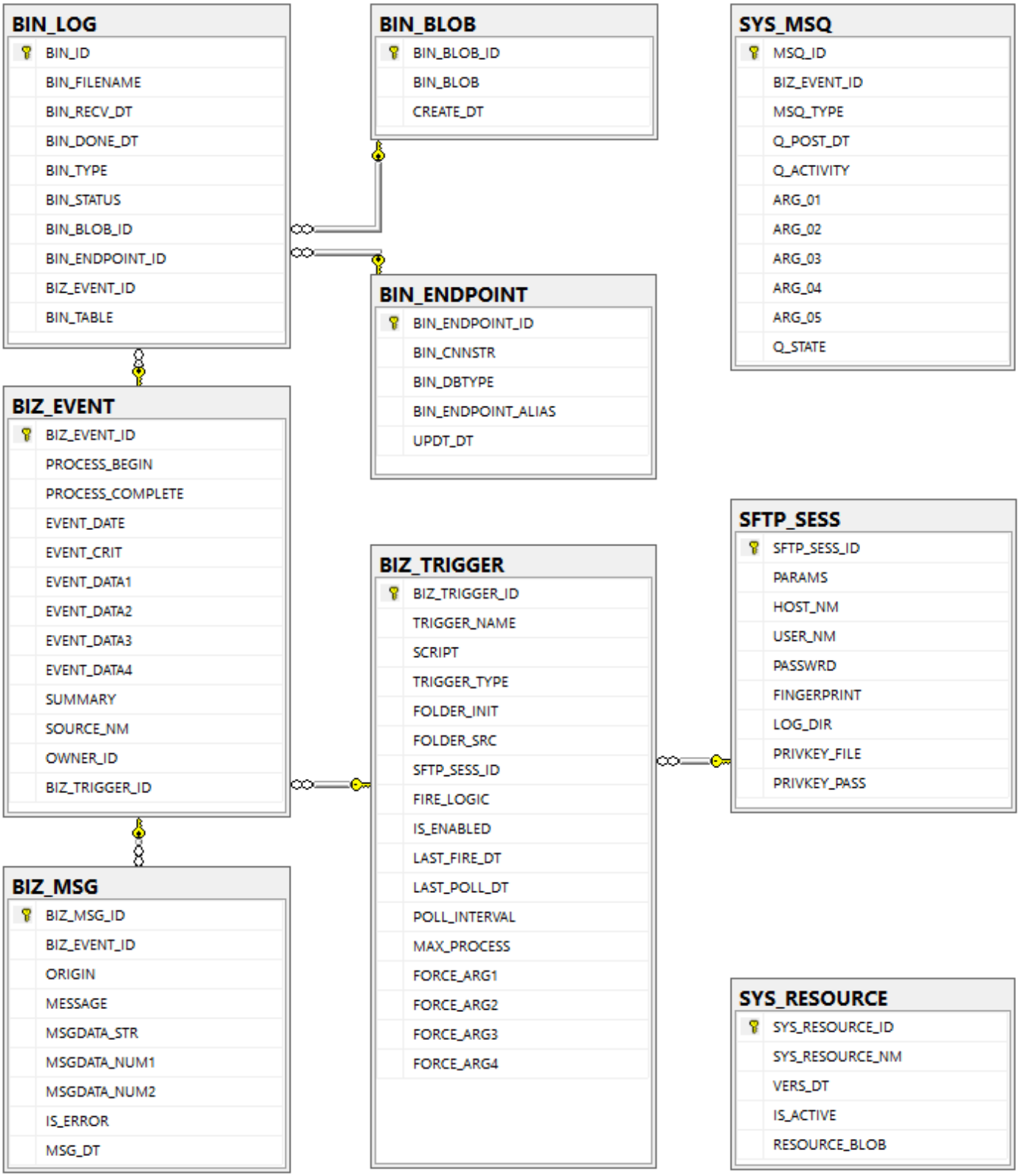
One of the design goals of these extracts is to convey these more complex EDI relationships; therefore, all of the above cases are present in the sample data and the seed file extract. Most of the “heavy lifting” for these requirements is handled in the large UNION subquery labeled MEM. This layer provides all the raw data used by the 2010BA, 2010BB, and 2010CA loops. Note that the HL segments are not mapped in this extract; SERENEDI dynamically generates the correct values for these segments based on the data being projected.

Within the clause itself, the UNION clause provides values for the Subscriber and Patient loops, depending on the PAR_MEMBER_ID column to decide if the member is a subscriber, and fills in or nulls out all values based on that information. This is one way to avoid needing excessive CASE WHEN … THEN … ELSE NULL END statements when encoding loops.

The final subquery generates Claim Charge Amounts and Claim From/To dates that are used at the 2300 Claim header loop level.

Finally, the ORDER BY statement at the end sorts data by Biller ID, Subscriber ID, Patient ID, Claim ID, and Claim Detail ID, ensuring the data is emitted in the correct order to create a valid file.

SERENEDI Architecture



On the previous page are the core tables for the serenediCore distribution database. Omitted from this diagram are the database objects related to the Sample Data and extracts, which are covered separately. These objects are used by the SERENEDI automation system, which is covered in deeper technical detail in this chapter.

As described in the introduction to SERENEDI, this platform was designed to facilitate portability, parallelism, and projections. Portability is made possible with the specific design choices and environment-related SCORE script commands that are part of the platform. Parallelism is possible due to the highly multi-process nature of the automation environment. Projections are provided as an aspect of the underlying technology.

SerenediService

When SerenediService is first run on either Windows or Unix, it reads the *cnnstr.txt* text file that defines the location of the serenediCore distribution database and pulls in information from the *license.key* file related to the licensing level of the SERENEDI installation.

SerenediService then sits in the background and runs the following checks at these intervals:

1. Four times a second, it polls the BIZ_EVENT table using the following query:

```
SELECT        BE.BIZ_TRIGGER_ID,
              MIN(BIZ_EVENT_ID) BIZ_EVENT_ID
FROM          BIZ_EVENT BE
LEFT JOIN
(
                    SELECT        BE.BIZ_TRIGGER_ID
                    FROM          BIZ_EVENT BE
                    INNER JOIN    BIZ_TRIGGER BT
                    ON            BT.BIZ_TRIGGER_ID = BE.BIZ_TRIGGER_ID
                    WHERE         PROCESS_BEGIN IS NOT NULL
                    AND           PROCESS_COMPLETE IS NULL
                    GROUP BY      BE.BIZ_TRIGGER_ID,
                                  BT.MAX_PROCESS
                    HAVING        COUNT(*) > COALESCE(MAX_PROCESS,9999)
) EXCEED
ON            EXCEED.BIZ_TRIGGER_ID = BE.BIZ_TRIGGER_ID
WHERE         BE.PROCESS_BEGIN IS NULL
AND           EXCEED.BIZ_TRIGGER_ID IS NULL
GROUP BY      BE.BIZ_TRIGGER_ID
ORDER BY      BIZ_EVENT_ID
```

   This query returns the oldest unprocessed event for each trigger that does not exceed the MAX_PROCESS count for concurrent processes, and sorts the results so that the oldest event from any trigger gets processed first. This query is re-run if there is worker capacity remaining.

   Maximum worker count is defined as either 7/8 of the available CPU cores or the licensed core count, whichever is less. Minimum worker count is defined as 1/4 of the maximum core count, with a minimum of 1. If the number of outstanding events exceeds the minimum worker count, new workers are launched up to the maximum worker count – otherwise, only the minimum worker count is maintained.

   If a worker executes an event that exceeds four hours of execution time, it is forcibly terminated and the event is marked TIMEOUT FAILURE in the SUMMARY column.

2. Four times a second, SerenediService checks the Trigger Scan process. If it is not running, it is relaunched.
3. Every 1.5 seconds, SerenediService checks for unprocessed data shuttle requests with this query:

   ```
   SELECT COUNT(*) FROM SYS_MSQ WHERE Q_STATE='U' AND MSQ_TYPE = 'DATA_SHUTTLE'
   ```

   If this value is nonzero and the data shuttle process is not running, it is relaunched.

4. Every 1.5 seconds, SerenediService checks to see if the SERENEDI Studio process is running. If not, and the serenediStudio.dll binary is found in the C:\serenedi\bin directory, it is relaunched.

SerenediService is designed to run in the background continuously – for this reason, it is solely concerned with launching child processes. The SerenediService process ID is communicated to all child processes; if SerenediService is terminated, all child processes will self-terminate as soon as possible.

On Windows, there are additional checks related to *memory resource exhaustion*. If the metrics detect insufficient RAM available, it will temporarily halt spawning new worker processes. These checks are not necessary on Unix.

Worker Process

Worker processes are orchestrated by SerenediService and operate in parallel with other worker processes. They are designed to be short-lived to safeguard against memory exhaustion. They exist to process a single, predetermined event, then exit.

If this time limit is exceeded before it finds work to do, it self-terminates. Four times a second, each worker process executes the following query:

```
SELECT TOP 1 BIZ_EVENT_ID FROM BIZ_EVENT WHERE PROCESS_BEGIN IS NULL AND OWNER_ID IS NULL
```

If this query returns a result, the worker next checks to see how many *other* worker processes are currently executing events tied to this same trigger. This is done because each trigger has a "parallelism throttle" defined in the MAX_PROCESS column that may limit the number of simultaneous worker processes servicing a single trigger. For example, a MAX_PROCESS of 1 means only one worker process could service events for that trigger at any given moment. If the throttle is exceeded, the worker continues to idle.

Next, the worker process attempts to take exclusive ownership of the event. If this ownership is successful, then the event is now "claimed" and ready to be processed. If the event shows a NULL value in the BIZ_TRIGGER_ID column, then it is an *immediate mode event* and it will treat the EVENT_DATA4 column as a local file location for a SCORE script to be executed. In this case, only the first three arguments are passed on to the script. If there is a trigger associated with this event, then the SCORE script associated with that event is executed and all four arguments are passed to the script.

Once the SCORE script has finished execution, the worker process immediately exits. Also, should the worker process exceed a total of five hours of execution time, the parent SerenediService will terminate it and mark the SUMMARY field for this event as TIMEOUT FAILURE.

Trigger Scan

The Trigger Scan first establishes a *time to live* between 1 and 11 minutes, and immediately starts to scan triggers with the following query:

```
SELECT * FROM BIZ_TRIGGER WHERE IS_ENABLED=1 AND POLL_INTERVAL > 0
```

Each row of the result set is processed as follows:

1. If the LAST_POLL_DT column is null or is set to a value that is older than the number of seconds stored in the POLL_INTERVAL column, then this trigger is eligible to be scanned.
2. The first thing to do is to update LAST_POLL_DT and add POLL_INTERVAL seconds to it. If the LAST_POLL_DT column is null, it is set to the current date/time.
3. If the TRIGGER_TYPE is set to LOCAL_UPLOAD, LOCAL_ARCHIVE, or SQL, control is passed to one of these handlers.

Upload Triggers

1. If an SQL filter is defined in the upload trigger, that is checked before anything else. In this way, developers can limit execution to certain days or times using standard SQL date/time expressions. If the SQL filter is not set, or if the SQL filter expression resolves to numeric 1, execution continues.
2. The Initial directory is defined in the FOLDER_INIT column. Upload triggers generate events when the Trigger Scan process is able to move files from the Initial directory to the Source directory defined in the FOLDER_SRC column. If file filters are set, they operate in the following way:
    a. When a single file filter is defined, this limits what files are scanned in the Initial directory. For example, a *.837 filter will ignore all files except those ending in the .837 filename extension.
    b. When *multiple* file filters are defined, they act to define file sets where the wildcard expressions must be matched on all filters, but only files matching on the first file filter will generate events.

        Example: Two filters are defined for this trigger: *.837 and *.HDR. Five files are present in the Initial directory: ABC.837, DEF.837, GHI.837, ABC.HDR, and DEF.HDR. This results in two files that can generate events: ABC.837 and DEF.837. The *.HDR files, not being the first file filter, do not generate events, but this filter does prevent GHI.837 from triggering an event because it does not form a complete file set.
3. If a TIMESTAMP filter is set on this event, it is used to evaluate a timestamp expression that is then added to the end of the filename when it is moved to a new location.
4. Files fulfilling all of the previously established criteria are moved to the directory as defined in the Source directory. If a file already exists in the Source directory, or if the file fails to be moved, it is assumed to be locked in some way and does *not* generate an event. Files are sorted based on their file size, with the largest files sorted first.
5. For all files satisfying the above criteria, an event is generated with the fully pathed filename as EVENT_ARG1 for each one.

Archive Triggers

1. If an SQL filter is defined in the archive trigger, that is checked first as defined in Step 1 for Upload triggers.
2. The sole directory used for Archive triggers is the Init directory defined in the FOLDER_INIT column of the trigger. All files present in the directory are scanned; if one or more file filters is set for this trigger, they are evaluated in exactly the same fashion as steps 2a and 2b defined for the Upload trigger above.
3. The bare filenames that result after step 2 is evaluated are compared with the results of this query:

```
SELECT EVENT_CRIT FROM BIZ_EVENT BE INNER JOIN BIZ_TRIGGER BT ON BE.BIZ_TRIGGER_ID =
BT.BIZ_TRIGGER_ID AND BT.IS_ENABLED=1 AND BT.BIZ_TRIGGER_ID=<<BIZ_TRIGGER_ID>>
```

4. Files that do *not* match are flagged as new and inserted as events. Files are sorted based on their file size, with the largest files sorted first. Within the event, the EVENT_CRIT column is the filename without paths, whereas EVENT_DATA1 is the fully pathed filename.

SQL Triggers

1. The SQL filter is the only thing checked for SQL triggers, which are basically yes/no flags to determine when events are generated.

Data Shuttle

The data shuttle is launched by SerenediService when there are unfulfilled data shuttle requests. The shuttle has a unique responsibility in the SERENEDI automation system to drive the BIN system, specifically flat and HDB data storage. Before the data shuttle can operate, the data needs to be “staged” properly so the shuttle can do its job. This means initially

sending the data to one or more temporary tables in the same database where the data will be stored. The automation system acts in slightly different ways depending on whether the data being stored is a single flat table or a multi-table HDB insertion.

When SERENEDI worker processes run SCORE scripts and execute the **sapi-FlatForceMergeToBIN** or **sapi-FlatMergeToBIN** commands, they perform these actions:

1. A new BIN ID is generated by the distribution database that uniquely identifies the data stored with this command.
2. A new record is created in the distribution database table SYS_MSQ as follows:
    a. Flat Merge operations: Q_ACTIVITY = 'FLAT_MERGE', Q_STATE = 'O'
    b. Flat ForceMerge operations: Q_ACTIVITY = 'FLAT_FORCEMERGE', Q_STATE = 'O'
    c. In both cases, the full database connection string, temp table name, and destination table name are stored in the SYS_MSQ record under the ARG_01 and ARG_02 columns. This is the information consumed by the data shuttle to transfer the information to its eventual destination.
    d. If the flag to suppress schema messages is set, ARG_03 will be set to NO_MSG.
3. The data is bulk stored to the destination SQL Server or Oracle database as a single temporary table with a table name beginning with **T_**<Event ID>**_**<10-digit random number. From the perspective of the database servers, these tables do not have any special designation as temporary tables.
4. After the table is successfully inserted, the SYS_MSQ entry for this table is changed from having a Q_STATE of O to a Q_STATE of U. This marks the table as ready for the data shuttle.

When SERENEDI worker processes run the SCORE script commands **sapi-HKeyMergeToHDB** and **sapi-HKeyForceMergeToHDB**, they commit these actions:

1. A new BIN ID is generated by the distribution database that uniquely identifies the data stored with this command.
2. Temp tables are generated in the distribution database for each loop in the source HKey register. They follow this naming convention: **T_**<Event ID>**_**<1-Based Increment>**_**<10-digit random number>. From the perspective of the database server, these tables have no specific classification as temporary tables.
3. Assuming that all the tables were inserted successfully, a single transaction inserts a SYS_MSQ for each table that was generated:
    a. HKey Merge operations: Q_ACTIVITY: 'FLAT_MERGE', Q_STATE='U'
    b. HKey ForceMerge operations: Q_ACTIVITY: 'FLAT_FORCEMERGE', Q_STATE='U'
    c. As above for the Flat commands, the full database connection string, temp table name, and destination table name are stored in the SYS_MSQ record under the ARG_01 and ARG_02 columns.
    d. If the flag to suppress schema messages is set, ARG_03 will be set to NO_MSG.

It may seem contrary to common sense to place all the data in temporary tables in the same database where the actual destination tables reside. What's the point? The reason is simple: parallelism. There are certain limitations in the databases themselves when inserting large amounts of data into a single table from possibly dozens of different processes. This is vastly compounded when the ForceMerge commands are used, and the destination table's schema has to expand dynamically to meet the needs of the incoming data. Both of these factors would severely limit the number of parallel data insert operations if no background data shuttle was involved.

Both of these factors become negligible with the data shuttle. For table merge shuttle requests, all the data shuttle needs to do is run an INSERT … SELECT statement and then a Drop Table command to fulfill the request, which is a very fast operation running in a single database. Since only one data shuttle process runs at any time, there's no danger of database Insert operations from multiple processes fighting for contention in a single table.

To handle Force Table shuttle requests, the data shuttle has more work to do. It treats schema alteration requests as expensive operations, and thus tries to aggregate schema alterations in all pending shuttle requests at once before processing any one request. In this way, the destination table schema is "prepped" to accept any or all of the data shuttle requests in a single operation, and then the insertions are handled exactly the same as normal merge requests.

The data shuttle will also generate messages in the BIZ_MSG if the NO_MSG flag in ARG_03 is not set. For Merge requests, data defined in columns in the source tables that are not present in the destination will result in messages with an Origin of FLAT_MERGE, Message of Unmerged Mapping, and the String Data field of the column name associated with the Event ID of the event making the request. When new columns are added, the Origin is FORCE_FLATMERGE, the message starts with COLUMN ADDED: and the column name, the Message String field is the destination table, and the message(s) are associated with the event making the request.

There is one implicit danger with the SERENEDI BIN system: the danger of cursors. When you open a cursor in a BIN table, you are also establishing a schema lock on the table, which then prevents the data shuttle from being able to fulfill ForceMerge schema requests. This could completely prevent the data shuttle from operating, leaving orphan temp tables or SYS_MSQ entries with a Q_STATE value of Z, which indicates a Flat insert failure. When you need to access the BIN tables, it is best to only do so with set-based operations. If you do need to iterate the data with a cursor, make a copy of the data with a set-based operation and then open the cursor in that copy – this will leave the BIN data completely free of interference.

BIN_LOG

This table is the key reference point for the entire BIN system – SERENEDI's method of storing a diverse set of information associated with EDI files. The BIN system can store Flat database stores and HDB (hierarchical database) stores. Each item committed is referenced with a single BIN_LOG_ID, often referred to as the BIN ID in this manual. The flat and hierarchical data stores must be accessible to normal SQL queries and can be stored locally in the serenediCore database or externally in another database. The location of the item is stored along with the BIN_LOG_ID reference so that when Data Fetch commands are executed, only the BIN_LOG_ID is needed to retrieve the item.

In most cases, the commands that store BIN items to the Flat and HDB stores do not usually *wait* for the item to be completely committed before the command concludes. That's because the data first immediately goes to a temp table in the serenediCore database before both the schema and data are *merged* to an existing data store in a way that is scalable across a large number of simultaneous transactions. The BIN_DONE_DT field is populated when the data for that BIN entry is fully committed and ready to be queried; while this field is null, you can assume that the data is *not* available to be read by your data consumer applications.

SERENEDI does not have any premade BIN tables for data to go to initially. Instead, when all default parameters are used, SERENEDI creates them dynamically as new EDI specifications are encountered.

Field Name	Data Type	Purpose
BIN_LOG_ID	**Integer (PK)**	This is the unique primary key for each BIN_LOG entry. It is the single point used to load in a BIN entry and is stored externally as a foreign key by all data stores.
BIN_FILENAME	**Varchar(200)**	This is the filename of the item that created the BIN entry.
BIN_RECV_DT	**DateTime (not null)**	This is the timestamp for when the BIN entry was created.
BIN_DONE_DT	**DateTime**	This is the timestamp for when the BIN entry data was committed to the database and became available for querying.

BIN_TYPE	**Integer (not null)**	102 – This entry references an HDB set of data tables. The BIN_TABLE will contain a *prefix* that goes before each of the Loop names. This increases both the difficulty of accessing the EDI data and the storage efficiency. 103 – This entry references a Flat data table. By default, it will reside in the serenediCore database and be named BIN_5010_837I (for a 5010 837 I file, in this example). This data is the least efficient for storage, but the easiest to access.
BIN_STATUS	**Varchar(20)**	**COMPLETE** – The BIN entry has completely finished processing. **PENDING** – The BIN entry is still being processed. **ERROR** – There was a critical error while processing the BIN entry.
BIN_BLOB_ID	**Integer (FK)**	This is only applicable to BIN_TYPEs 100 and 101 (SegPool and XML files) and is a foreign key to the BIN_BLOB table, where the raw data for this entry is stored.
BIN_ENDPOINT_ID	**Integer (FK)**	This is an optional foreign key to the BIN_ENDPOINT table. When present, it signifies that the BIN entry is residing outside the default serenediCore database in a different database or server.
BIZ_EVENT_ID	**Integer (FK)**	When populated, this indicates the BIZ_EVENT_ID foreign key of the event that created this entry.
BIN_TABLE	**Varchar(200)**	This is mandatory for HDB and Flat entries and is the data table name where the BIN entry is stored.

BIN_BLOB

This table is reserved for future use.

Field Name	Data Type	Purpose
BIN_BLOB_ID	**Integer (PK)**	This is the primary key of the table, referenced in the BIN_LOG table.
BIN_BLOB	**Varbinary (max)**	This column is the actual storage of the binary object.
CREATE_DT	**DateTime (not null)**	This is the creation date of the BLOB.

BIN_ENDPOINT

This table stores references to other databases and database servers. Furthermore, it sets up “aliases” that can be used during SCORE workflow steps as shorthand for storing EDI-related objects in the BIN system.

Field Name	Data Type	Purpose
BIN_ENDPOINT_ID	**Integer (PK)**	This is the primary key to the table.
BIN_CNNSTR	**Varchar(2000) (not null)**	This is the Connection String used for opening the connection. The exact format of the string depends on the database provider used.
BIN_DBTYPE	**Varchar(20) (not null)**	This represents the database server used for the database. **SQLSERVER –** MS SQL Server (version 2012 or above is supported)
BIN_ENDPOINT_ALIAS	**Varchar(100)**	This is the alias used for this database connection.
UPDT_DT	**DateTime (not null)**	This is the creation date or update time of the connection.

BIZ_TRIGGER TABLE

This is the core table that drives the generation of events from the SERENEDI automation system. See the "Events" chapter for more information about how the values in the table's fields drive the generation of new events.

Field Name	Data Type	Purpose
BIZ_TRIGGER_ID	**Integer (PK)**	This is the primary key for the table. It is auto-generated upon record insertion.
TRIGGER_NAME	**Varchar(200)**	This is the name of the trigger. If the trigger is grouped with other triggers as part of a business process, they should share a common prefix to make it clear that the process is associated with a group.
SCRIPT	**Varchar(200)**	This is the path and filename to the SCORE script executed by the events generated by this trigger.
TRIGGER_TYPE	**Varchar(20)**	This is defined as follows: **SQL**: An SQL query is executed during the trigger polling; a 1 integer result from the SQL fires the event. The SQL is defined in the FIRE_LOGIC column. **PASSIVE**: The trigger will not actively fire events, but is used to link events to a SCORE script. **LOCAL_UPLOAD**: This trigger will create new events for every file that is 1) Placed in the FOLDER_INIT folder and 2) can be successfully moved to the FOLDER_SRC folder. **LOCAL_ARCHIVE**: This trigger fires for every file that is present in the file system but *not* present as an event in the BIZ_EVENT table.
FOLDER_INIT	**Varchar(200)**	This is used for both LOCAL_UPLOAD and LOCAL_ARCHIVE triggers. For LOCAL_UPLOAD triggers, the file starts here and then is moved to the FOLDER_SRC. For LOCAL_ARCHIVE triggers, only this folder is used.
FOLDER_SRC	**Varchar(200)**	This is used by the LOCAL_UPLOAD triggers as the destination to move files *to* prior to firing the trigger. Syntax in the FIRE_LOGIC column is available to post-fix a timestamp and give a unique identifier to files that have the same name.
SFTP_SESS_ID	**Integer (FK)**	This is a foreign key to the SFTP Session table, used to define a passive SFTP Mirror trigger. These triggers do not spawn any events of their own – instead, they trigger SFTP Mirror operations via the pipeline system. The *local* directory is specified in FORCE_ARG3, and the *remote* directory is specified in FORCE_ARG4. The LAST_POLL_DT and POLL_INTERVAL, in turn, are used to track when the SFTP server is polled.
FIRE_LOGIC	**Varchar(4000)**	This is a pipe-delimited list of filters. TIMESTAMP applies only to UPLOAD trigger types. SQL is used for all triggers, and FILTER is used only for UPLOAD and ARCHIVE triggers. More information is available in the "Triggers" chapter.
IS_ENABLED	**Integer**	When this value is 1, the trigger is enabled. Any other value will disable the trigger.
LAST_FIRE_DT	**DateTime**	This defines the last time the trigger fired to create events.
LAST_POLL_DT	**DateTime**	The defines the last time the trigger was polled.
POLL_INTERVAL	**Integer (not null)**	This is the number of seconds between polls. Setting this to a value below 1 completely disables the trigger.
MAX_PROCESS	**Integer**	This is the limit on the number of active events that can process a trigger at the same time. Setting it to 1 ensures that any events created by this trigger will be serial – the next event will not be fired until the previous one is completely finished.
FORCE_ARG1	**Varchar(200)**	

FORCE_ARG2	**Varchar(200)**	When specified, these values will set the EVENT_ARG1 to EVENT_ARG4 columns in the newly created trigger to a fixed value. This is primarily useful when you'd like to create a single SCORE script that handles multiple triggers or business processes. By forcing a fixed value into one of the arguments, the SCORE script can branch execution and assign dedicated handlers to each process based on the argument. In this way, a SCORE script is able to orchestrate a full business system as opposed to handling an individual functional process. For SQL triggers, FORCE_ARG4 will be used as a BIN Endpoint Alias to allow the SQL defined in the FIRE_LOGIC column to be executed in external databases.
FORCE_ARG3	**Varchar(200)**	
FORCE_ARG4	**Varchar(200)**	

BIZ_EVENT TABLE

One row in the BIZ_EVENT table represents one unit of work that is scheduled to be farmed out to a number of parallel worker processes. Most of the time, this unit of work is centered around a single file moving through the EDI pipeline via the Upload or Archive triggers, but the system is flexible enough to do many other things, triggered actively via SQL triggers or passively via external inserts to this table.

Events are "owned" by worker processes as described in more detail earlier in this section, carrying specific arguments to the SCORE scripts associated with the trigger that generated the event. When large groups of events are triggered by a large number of files being processed at once, SERENEDI will attempt to process the largest files first. This helps reduce the overall time to complete the workload.

This system is also flexible enough to operate without an associated trigger – if the BIZ_TRIGGER_ID value is null, the EVENT_DATA4 column is treated as the local filename of a SCORE script to execute. This feature enables new SCORE scripts to bootstrap and prepare the local environment.

The Event Date column indicates when the event was created, the Process Begin column shows when a worker process started work on the event, and Process Complete indicates when the worker process was completed. If the event is completed without critical errors, it is flagged as SUCCESS in the Summary column.

If you are having difficulty troubleshooting an event that went wrong, it's possible to use the Visual Code (with Power Shell Core version 6 extensions) debugging environment to "replay" the event line by line. Instructions for doing this are found at the end of the Visual Code installation instructions earlier in the manual. The **Set-Location** and **sapi-InitializeSession** commands both need to point to the binary and base directories of the SERENEDI installation:

```
Set-Location C:\serenedi\bin
Import-Module -Name (Resolve-Path 'SERENEDI2.dll')
sapi-InitializeSession -BaseDir 'C:\serenedi' -BizEventId <<Event ID here>>
```

The third command will load the event-specific data into the SERENEDI runtime environment, and then the SCORE script can be executed line by line using a debugger.

Field Name	Data Type	Purpose
BIZ_EVENT_ID	**Integer (PK)**	This the unique primary key of the EVENT. It is referenced by the BIZ_MSG table, the SYS_MSG table, and the BIN_LOG table.
PROCESS_BEGIN	**DateTime**	This is the timestamp for when a worker process took ownership of this event and commenced work on it.

PROCESS_COMPLETE	**DateTime**	This is the timestamp for when the worker process completed the event.
EVENT_DATE	**DateTime**	This is the timestamp for when the event was created.
EVENT_CRIT	**Varchar(4000)**	This is the event *criteria*, generally the filename or some other string that fired the triggering mechanism.
EVENT_DATA1	**Varchar(1000)**	These event *data* fields provide data about the feeds to the SCORE script linked to the trigger that created this event. Alternatively, if the event is spawned with a *null* BIZ_TRIGGER_ID, then SERENEDI will check the field EVENT_DATA4 if it references a file. If so, then it's assumed this is a SCORE script executed in immediate mode, which does not require a trigger, and EVENT_DATA1, 2, and 3 are passed to this script during execution.
EVENT_DATA2	**Varchar(1000)**	
EVENT_DATA3	**Varchar(1000)**	
EVENT_DATA4	**Varchar(1000)**	
SUMMARY	**Varchar(200)**	This is generally either SUCCESS when an event completes without errors, or CRITICAL FAILURE if an error occurred when executing the SCORE script for this event.
SOURCE_NM	**Varchar(1000)**	This indicates the type of trigger that created the event. The three types are LOCAL_UPLOAD, LOCAL_ARCHIVE, and SQL.
BIZ_TRIGGER_ID	**Integer**	This is a foreign key reference to an item in the BIZ_TRIGGER table. If it is not supplied, then the event is executed in *immediate* mode.

BIZ_MSG

This table stores all messages generated in the course of processing events. Messages are not inserted into the table as they occur; instead, once the SCORE script for an event is completed, the messages are inserted in the order they were generated and tied to the BIZ_EVENT_ID.

Field Name	Data Type	Purpose
BIZ_MSG_ID	**Integer (PK)**	This is the unique primary key for the message.
BIZ_EVENT_ID	**Integer (not null)**	This is the foreign key to the BIZ_EVENT_ID that spawned this message.
ORIGIN	**Varchar(50)**	This is a short string that denotes the origin of the message. Possible values are: **DATA_SHUTTLE** – This denotes fields that were added by the BIN system or flagged as not present. **USER** – This indicates a message generated by a SCORE script and given the default ORIGIN. For the syntax error messages, each "family" of errors will have a different Origin. The SCORE command documentation will more thoroughly cover those errors.
MESSAGE	**Varchar(400)**	This is the primary point of communication for the message. It does not need to be too specific; it can rely on the following data fields to add information about what triggered this message.
MSGDATA_STR	**Varchar(400)**	This is string data for the message.
MSGDATA_NUM1	**Integer**	This is the first integer data for the message. If the message is a segment syntax error, this will store the segment index within the file of where the error occurred.
MSGDATA_NUM2	**Integer**	This is the second integer data for the message.
IS_ERROR	**Integer**	This is a flag, either 1 or 0, that indicates whether this message should be considered an error.
MSG_DT	**DateTime (not null)**	This is the timestamp for when the message was created.

SFTP_SESS

This table stores everything about a SecureFTP session, including sites, usernames, and passwords. It is used by the SFTP commands in the SCORE scripting system to centrally warehouse all the information related to SFTP sessions. If a fingerprint is received from an SFTP server that does not match a value set in this table, the command will fail with an error message.

Field Name	Data Type	Purpose
SFTP_SESS_ID	**Integer (PK)**	This is the unique primary key to the SFTP session.
PARAMS	**Varchar(2000)**	This is a comma-delimited list of options that direct the behavior of the SFTP session. They are: **BINARY** – Enforces a binary transfer mode during file operations. **ASCII** – Enforces an ASCII transfer mode during file operations. **LOCAL_MIRROR** – For Directory Mirror operations, this option directs new remote files to be downloaded to the local file system. If neither LOCAL_MIRROR nor REMOTE_MIRROR is supplied, this is the default mode of operation. **REMOTE_MIRROR** – For Directory Mirror operations, this option overrides the default Local Mirror setting and uploads files found on the local folder to the remote folder. **FILE_MOVE** – This option removes the source file after it is successfully uploaded or downloaded to the destination system.
HOST_NM	**Varchar(200) (not null)**	This is either the IP address or host name of the SFTP Site.
USER_NM	**Varchar(100) (not null)**	This is the username credential for the session.
PASSWRD	**Varchar(100) (not null)**	This is the password for the session.
FINGERPRINT	**Varchar(200)**	This is the remote site *fingerprint*. It is set automatically when first connecting to a remote SFTP site. Any subsequent connection will require the same fingerprint – if any changes are made, the connection will not go through and an error message will be logged. If you need to reset the fingerprint, it can be manually set to the string VOID. The next time this SFTP connects to the remote server, the fingerprint will be regenerated from the remote site.
LOG_DIR	**Varchar(200)**	This specifies a local file system directory that will be used to generate plain-text logs of SFTP operations as they occur.

PRIVKEY_FILE	**Varchar(200)**	Certain SFTP servers require public/private keys. This file refers to the local client's private key file. To be considered valid by the SSH.NET library used for SFTP communications, it must be in plain text and start and end with the following strings:
		`-----BEGIN RSA PRIVATE KEY-----`
		`-----END RSA PRIVATE KEY-----`

PRIVKEY_PASS	**Varchar(200)**	If a passphrase is needed to unlock the Private Key file, supply it in this column.

SYS_MSQ

The function of the SYS_MSQ table is explained earlier, in the SERENEDI Architecture section about the data shuttle. It is a temporary workspace that enables the background data shuttle service to complete data storage requests. When the data

storage request is successfully completed, the SYS_MSQ row is deleted. Only in the event of a critical failure will the row be left behind with a Q_STATE status of Z. If the row is stuck in a state of O, it means the Flat data could not be stored to the destination database as a temp table. If the row is stuck in a state of U, either the data shuttle is not running or it cannot access the distribution database.

When no data shuttle requests are ongoing, this table should have no rows, and letting Z error records accumulate in this table could slow down the overall performance of the system.

Field Name	Data Type	Purpose
MSQ_ID	**Integer (PK)**	This is the unique primary key of the MSQ record.
BIZ_EVENT_ID	**Integer (FK)**	This links the request to the event that spawned the data request. If a problem occurs, you can use this to analyze the root cause.
MSQ_TYPE	**Varchar(20)**	This value is hard-coded as DATA_SHUTTLE. It is open to future expansion.
Q_POST_DT	**DateTime**	If there is a critical error, this is the time the record errored out.
Q_ACTIVITY	**Varchar(50)**	This has two possible values: FLAT_MERGE FORCE_FLATMERGE The first value is a request to merge the data source in ARG_01 to the destination in ARG_02 without making schema changes. If the NO_MSG flag is not set in ARG_03, it will create messages for any fields in the source it can't find in the destination. The second, FORC_FLATMERGE, is a request to merge the data source to the destination with schema changes. This way, all the data in the source can be inserted into the destination.
ARG_01	**Varchar(1000)**	This specifies the *source* of the shuttle request, in this format: Flat SQL Server requests: SQLSERVER, Connection String, Temp Table Name Flat Oracle requests: ORACLE, Connection String, Temp Table Name HDB SQL Server requests: SQLSERVER, Connection String, Temp Table Name Prefix_LoopShortName HDB Oracle requests: ORACLE, Connection String, Temp Table Name Prefix_LoopShortName
ARG_02	**Varchar(1000)**	This specifies the *destination* of the shuttle request. The format is the same as indicated in ARG_01. The values are the same as the source of the database type and connection string since the job of the data shuttle is to merge two data tables that are already in the same database. Instead of the Temp Table Name, however, for Flats, it will end with the destination data table, and for HDB shuttle requests, it will contain Prefix_LoopShortName.
ARG_03	**Varchar(1000)**	If the SCORE script requests that no schema messages are generated for Merge or ForceMerge operations, this should be set to NO_MSG. If it is not set, schema messages will be inserted into the BIZ_MSG table and associated with the event.

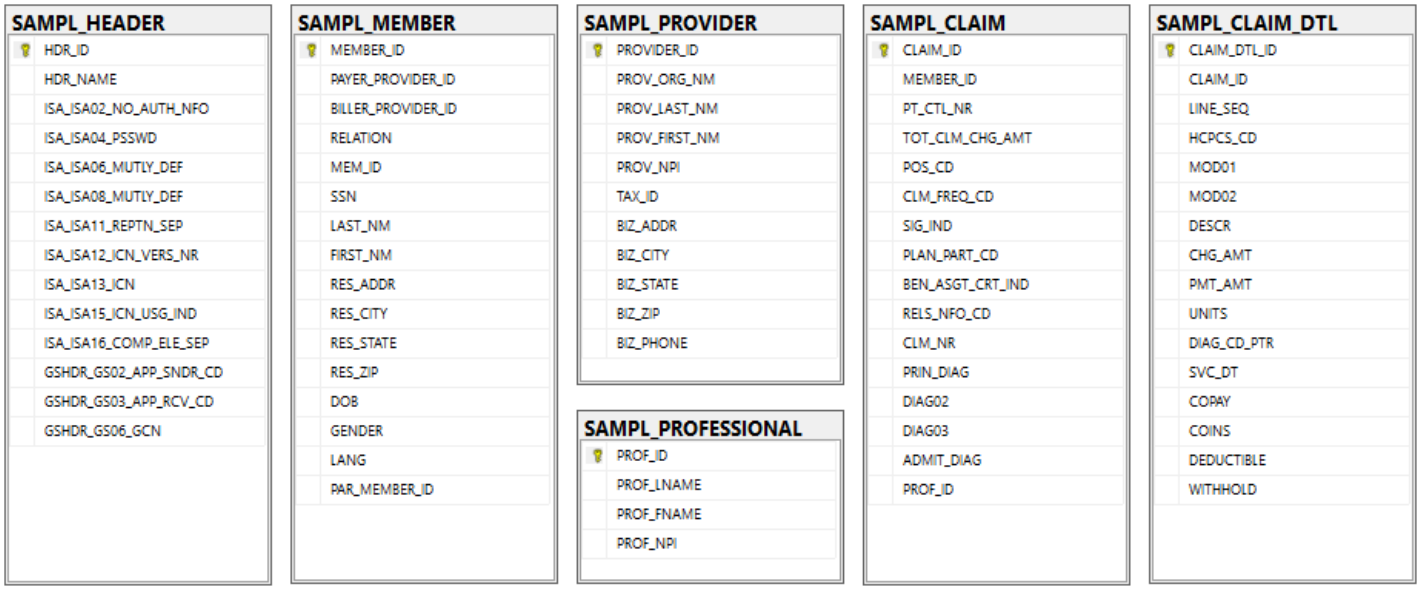
ARG_04	**Varchar(1000)**	This is used for HDB data shuttle requests and contains the two-digit Tree Specification Tag. While Flat tables contain the Tree Specification Tag within the first mapping, Hierarchical Data Tables contain it only in the table corresponding to the ISA loop. This allows the data shuttle to correctly determine the mapping space and data types.
ARG_05	**Varchar(1000)**	Reserved for future use.
Q_STATE	**Varchar(1)**	This is a single-character status code: O – A temp Flat transfer is beginning but not finished U – HDB/Flat data table is ready for transfer to the destination Z – A critical error occurred while processing this request

SYS_RESOURCE

This table contains binary resources utilized by the SERENEDI engine. It is not user-accessible.

Field Name	Data Type	Purpose
SYS_RESOURCE_ID	**Integer (PK)**	Primary key to the resource table
SYS_RESOURCE_NM	**Varchar(100) (not null)**	Name of the resource
VERS_DT	**DateTime (not null)**	Timestamp of the resource version
IS_ACTIVE	**Integer (not null)**	Normally 1
RESOURCE_BLOB	**Varbinary (max) (not null)**	Binary value of the resource

Sample Data Tables



The Sample Data Tables illustrated above store mock data related to claims, members, and providers so they can provide the raw data needed to create the seed files in the sample extracts. These are covered further in the “Creating Outbound Transactions” chapter.

SAMPL_CLAIM

This table represents 337 claims tied to both members and subscribers, allowing the 837 extracts to illustrate more complex relationships between subscribers and members when encoding the claims. This table is also used for encoding 835 files.

Field Name	Data Type	Purpose
CLAIM_ID	**Integer (PK)**	Primary key to the Sample Claim table
MEMBER_ID	**Integer**	Foreign key to the SAMPL_MEMBER table
PT_CTL_NR	**Varchar(300)**	Patient control number
TOT_CLM_CHG_AMT	**Numeric(18,2)**	Total claim charge amount
POS_CD	**Varchar(300)**	Place of service code
CLM_FREQ_CD	**Varchar(300)**	Claim frequency code
SIG_IND	**Varchar(300)**	Signature indicator
PLAN_PART_CD	**Varchar(300)**	Plan participation code
BEN_ASGT_CRT_IND	**Varchar(300)**	Benefits assignment indicator
RLS_NFO_CD	**Varchar(300)**	Release of information code
CLM_NR	**Varchar(300)**	Claim number
PRIN_DIAG	**Varchar(300)**	Principal diagnosis
DIAG02	**Varchar(300)**	Secondary diagnosis
DIAG03	**Varchar(300)**	Tertiary diagnosis
ADMIT_DT	**Varchar(300)**	Admission date
PROF_ID	**Integer**	Foreign key to the SAMPL_PROFESSIONAL table

SAMPL_CLAIM_DTL

This table contains claim lines tied to the claims. For encoding to 835 files, the patient responsibility amounts (copay, coinsurance, deductible, withholding) are used for encoding CAS adjustment segments.

Field Name	Data Type	Purpose
CLAIM_DTL_ID	**Integer (PK)**	Primary key to the Claim Detail table
CLAIM_ID	**Integer**	Foreign key to the SAMPL_CLAIM table
LINE_SEQ	**Integer**	Line sequence
HCPCS_CD	**Varchar(300)**	Procedure code
MOD01	**Varchar(300)**	Procedure modifier 01
MOD02	**Varchar(300)**	Procedure modifier 02
DESCR	**Varchar(300)**	Description
CHG_AMT	**Numeric(18,2)**	Line charge amount
PMT_AMT	**Numeric(18,2)**	Line payment amount
UNITS	**Numeric(18,2)**	Units
DIAG_CD_PTR	**Varchar(300)**	Diagnosis code pointer
SVC_DT	**DateTime**	Service date
COPAY	**Numeric(18,2)**	Patient copay
COINS	**Numeric(18,2)**	Patient coinsurance
DEDUCTIBLE	**Numeric(18,2)**	Patient deductible
WITHHOLD	**Numeric(18,2)**	Patient withholding

SAMPL_HEADER

This contains sample data used in the outer envelopes of all the extracts.

Field Name	Data Type	Purpose
HDR_ID	**Integer (PK)**	Primary key to the Header table
HDR_NAME	**Varchar(300)**	Header name
ISA_ISA02_NO_AUTH_NFO	**Varchar(300)**	ISA envelope map
ISA_ISA04_PASSWD	**Varchar(300)**	ISA envelope map
ISA_ISA06_MUTLY_DEF	**Varchar(300)**	ISA envelope map
ISA_ISA08_MUTLY_DEF	**Varchar(300)**	ISA envelope map
ISA_ISA11_REPTN_SEP	**Varchar(300)**	ISA envelope map
ISA_ISA12_ICN_VERS_NR	**Varchar(300)**	ISA envelope map
ISA_ISA13_ICN	**Integer**	ISA interchange control number
ISA_ISA15_ICN_USG_IND	**Varchar(300)**	ISA envelope map
ISA_ISA16_COMP_ELE_SEP	**Varchar(300)**	ISA envelope map
GSHDR_GS02_APP_SNDR_CD	**Varchar(300)**	GS envelope map
GSHDR_GS03_APP_RCV_CD	**Varchar(300)**	GS envelope map
GSHDR_GS06_GCN	**Integer**	GS control number

SAMPL_MEMBER

This table defines 170 members/subscribers using completely random information. The relation code is D for dependent (member) or P for primary (subscriber). Dependents are tied to the parent record via the PAR_MEMBER_ID column.

Field Name	Data Type	Purpose
MEMBER_ID	**Integer (PK)**	Primary key to the Member table
PAYER_PROVIDER_ID	**Integer**	Foreign key to the SAMPL_PROVIDER table
BILLER_PROVIDER_ID	**Integer**	Foreign key to the SAMPL_PROVIDER table
RELATION	**Varchar(300)**	Relation code
MEM_ID	**Varchar(300)**	Plan member identification code
SSN	**Varchar(300)**	Social Security number
LAST_NM	**Varchar(300)**	Last name
FIRST_NM	**Varchar(300)**	First name
RES_ADDR	**Varchar(300)**	Residential address
RES_CITY	**Varchar(300)**	Residential city
RES_STATE	**Varchar(300)**	Residential state
RES_ZIP	**Varchar(300)**	Residential ZIP
DOB	**Date**	Date of birth
GENDER	**Varchar(300)**	Gender
LANG	**Varchar(300)**	Language
PAR_MEMBER_ID	**Integer**	Foreign key to SAMPL_MEMBER table (Parent record)

SAMPL_PROFESSIONAL

This table is a simple list of six professional providers. Their last names are all types of rocks.

Field Name	Data Type	Purpose
PROF_ID	**Integer (PK)**	Primary key to the Professional Table

PROF_LNAME	**Varchar(300)**	Last name
PROF_FNAME	**Varchar(300)**	First name
PROF_NPI	**Varchar(300)**	NPI

SAMPL_PROVIDER

This table contains six business-level entity providers.

Field Name	Data Type	Purpose
PROVIDER_ID	**Integer (PK)**	Primary key to the Provider Table
PROV_ORG_NM	**Varchar(300)**	Organization name
PROV_LAST_NM	**Varchar(300)**	Last name
PROV_FIRST_NM	**Varchar(300)**	First name
PROV_NPI	**Varchar(300)**	NPI
TAX_ID	**Varchar(300)**	Tax ID
BIZ_ADDR	**Varchar(300)**	Address
BIZ_CITY	**Varchar(300)**	City
BIZ_STATE	**Varchar(300)**	State
BIZ_ZIP	**Varchar(300)**	ZIP
BIZ_PHONE	**Varchar(300)**	Phone

Appendix A: SerenediAPI Workflow Reference

Serenedi is driven by PowerShell Core cmdlets. This cross-platform scripting solution is based on .NET Core and has a track record going back many years. PowerShell Core is easily extensible with external libraries, which makes it an ideal scripting solution for an integration platform.

Global Variables

When an event is fired, these variables will be populated prior to running the script:

Variable Name	Purpose
folderSrc	Source Folder
eventCrit	Event Trigger Criteria
eventData1	Argument Data 1
eventData2	Argument Data 2
eventData3	Argument Data 3
eventData4	Argument Data 4
mainScript	Main Process Execution Script

sapi-FetchBinState	This command is used to interrogate the state of a BIN item. It provides the type of the BIN item and tells whether it is ready for use. If the BIN has not finished writing, one of these values will be returned: HKEY_PEND FLAT_PEND UNKNOWN_PEND If the BIN has completed being stored, one of these values will be returned, depending on the type of the BIN item: FLAT HKEY UNKNOWN
Parameter	**Value**
-BinId (int, mandatory)	Provide the BIN ID of the item to be investigated.

sapi-FlatForceMergeToBIN	This command will merge the loaded Flat table to the BIN system. If mappings are present in the Flat that are not in the destination BIN schema, the columns will automatically be added to the destination schema. Messages for these new files will be added to the message log *unless* the SuppressSchemaMsg flag is set. The return value from this command is an **int** of the BIN ID.
ERRORS	
WFDS0010	DataShuttle service not processing queued request – 20 minute timeout
WFDS0020	DataShuttle critical error while servicing request
WFDS0030	Critical error creating temp table
Parameter	**Value**
-Table (string, optional)	When specified, this will override the default destination BIN table with a provided table name.
-Filename (string, optional)	When provided, this will be placed in the BIN_FILENAME column of the new BIN entry.

-SupressSchemaMsg (boolean, optional)	If set to True, SERENEDI will not generate message logs for new columns that are not present in the destination Flat schema.
-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.
-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
-NoWait (boolean, optional)	The default behavior of this cmdlet is to wait for the time it normally takes for the data to be inserted into the destination Flat table by the background data shuttle process. If this value is set to True, the script will continue execution.

sapi-FlatFromBIN	This is a general database command to load the Flat register. If a BIN ID is supplied, the Flat will be retrieved from the BIN system. If a table is supplied, the SQL in the table parameter will be executed and the Flat will be retrieved from that result instead. Note that "table" here can also mean an SQL View or an SQL Stored Procedure with parameters. When creating ORACLE Stored Procedures, the first argument in the procedure must be "PRC OUT SYS_REFCURSOR". SERENEDI will use this cursor to fetch results from the stored procedure. If one or both of the string arguments are used, they must be named "ARG1" and "ARG2", respectively.
ERRORS	
BINX0085	Referenced BIN ID does not exist in the BIN_LOG table.
BINX0080	Cannot load Flat from this BIN ID: It is not storage in FLAT format.
BINX0120	Critical error during Fetch_BIN_Flat. Make sure that the SQL returns a valid dataset.
Parameter	**Value**
-BinId (int, optional)	When specified, this will direct SERENEDI to load the Flat register from the BIN system for the specified BIN ID.
-Table (string, optional)	This is mandatory if the BIN ID is not provided. The SQL given in this argument will be executed and the result will be processed into the Flat register. If this is prefixed with the capital letters EXEC, then the results of the following *stored procedure* will be assigned to the Flat register.

-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.
-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
-Arg1 (string, optional)	This is an optional arguments that can be used when calling on stored procedures.
-Arg2 (string, optional)	This is an optional arguments that can be used when calling on stored procedures.

sapi-FlatMergeToBIN	This command will merge the loaded Flat table to the BIN system. If there are mappings present in the Flat that are not in the destination BIN schema, messages will be generated *unless* the SuppressSchemaMsg flag is set. The return value from this command is an **int** of the BIN ID.
ERRORS	
WFDS0010	DataShuttle service not processing queued request – 20 minute timeout
WFDS0020	DataShuttle critical error while servicing request
WFDS0030	Critical error creating temp table
Parameter	**Value**
-Table (string, optional)	When specified, this will override the default destination BIN table with a provided table name.
-Filename (string, optional)	When provided, this will be stored in the BIN_FILENAME column of the new BIN entry.
-SupressSchemaMsg (boolean, optional)	If set to True, SERENEDI will not generate message logs for new columns that are not present in the destination Flat schema.
-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.
-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
-NoWait (boolean, optional)	The default behavior of this cmdlet is to wait for the time it normally takes for the data to be inserted into the destination Flat table by the background service process. If this value is set to True,

	the default behavior will be overridden and the script will continue execution.

sapi-HKeyMergeToHDB	This command will merge the loaded HKey register to an existing HDB tableset. If the destination HDB tableset does not have column mappings that are present in the HKey, a message will be generated once for that mapping *unless* the SuppressSchemaMsg flag is set. The return value from this command is an **int** of the BIN ID.
ERRORS	
WFHDS0010	DataShuttle critical error while creating table
WFHDS0020	DataShuttle critical error while servicing request
WFHDS0030	DataShuttle service not processing queued request – 20 minute timeout
Parameter	**Value**
-Prefix (string, optional)	If supplied, this will override the default HDB_5010_<Specification Name> prefix for the HDB tableset.
-Filename (string, optional)	If supplied, the filename will be provided for this BIN entry.
-SupressSchemaMsg (boolean, optional)	If set to True, SERENEDI will not generate message logs for columns that can't be stored in the existing HDB schema.
-BinEndpointId (int, optional)	If supplied, this will provide the Endpoint ID for the destination database connection.
-BinEndpointAlias (string, optional)	If supplied, this will provide an Endpoint Alias for the destination database connection.
-NoWait (boolean, optional)	The default behavior of this cmdlet is to wait for the time it normally takes for the data to be inserted into the destination HDB tables by the background service process. If this value is set to True, the default behavior will be overridden and the script will continue execution.

sapi-HKeyForceMergeToHDB	This command will merge the loaded HKey register to an existing HDB tableset. If the destination HDB tableset does not have column mappings that are present in the HKey, new mappings will be created and a message will be generated to that effect *unless* the SuppressSchemaMsg flag is set. The return value from this command is an **int** of the BIN ID.
ERRORS	
WFHDS0010	DataShuttle critical error while creating table
WFHDS0020	DataShuttle critical error while servicing request
WFHDS0030	DataShuttle service not processing queued request – 20 minute timeout
Parameter	**Value**
-Prefix (string, optional)	If supplied, this will override the default HDB_5010_<Specification Name> prefix for the HDB tableset.
-Filename (string, optional)	If supplied, the filename will be provided for this BIN entry.
-SupressSchemaMsg (boolean, optional)	If set to True, SERENEDI will not generate message logs for new columns that will be added to the destination HDB schema.
-BinEndpointId (int, optional)	If supplied, this will provide the Endpoint ID for the destination database connection.
-BinEndpointAlias (string, optional)	If supplied, this will provide an Endpoint Alias for the destination database connection.
-NoWait (boolean, optional)	The default behavior of this cmdlet is to wait for the time it normally takes for the data to be inserted into the destination HDB tables by the background service process. If this value is set to True, the default behavior will be overridden and the script will continue execution.

sapi-HKeyFromHDB	This command will fetch data from an existing HDB BIN ID and load it into the HKey register. The database connection used will be the one that was used when the BIN ID was generated.
ERRORS	
BINX0070	No entry with this BIN_ID exists in the database.
BINX0080	The ISA base table was not found for this BIN
BINX0090	Critical error during Fetch_BIN_HKey

Parameter	Value
-BinId (integer, mandatory)	The BIN ID of the HDB tableset present in the BIN system.

CSV COMMANDS

sapi-CSVToDB	This will load a CSV file and commit it to a BIN endpoint. The CSV file does *not* need to be CGIF2 formatted, but must contain only string data. The first row must contain the column definitions and the CSV must contain the same columns as a specified schema table. If any columns are missing, extra, or out of order, an error result will trigger. This command gives SERENEDI the ability to import general CSV data from an external source. The destination table and schema table need to have an integer column called BIN_ID, which will be populated during transfer automatically from the BIN system. The source CSV should *not* have this column.
ERRORS	
CSV2DB0010	Unable to open DB Connection **STR:** filename
CSV2DB0020	Error Processing CSV To Table **STR:** column
CSV2DB0030	Error Processing CSV To Table – COLUMN NOT FOUND IN DESTINATION SCHEMA AT EXPECTED POSITION **STR:** column
CSV2DB0040	Error Processing CSV To Table – COLUMN MISMATCH – DESTINATION SCHEMA CONTAINS DIFFERENT COLUMN COUNT TO DESTINATION SCHEMA **N1:** destination column count **N2:** source column count
CSV2DB0050	Error Processing CSV To Table – ERROR BULK LOADING FILE **STR:** Exception message
CSV2DB0060	Error Processing CSV To Table – BIN_ID COLUMN FOUND BUT NOT EXPECTED **STR:** column
CSV2DB0070	General error sending CSV to DB **STR:** Exception message
Parameter	**Value**

-Filename (string, mandatory)	This is the filename of the headered CSV to load. It should contain only string quote–delimited data and have a single row header defining the columns.
-SchemaTable (string, mandatory)	This is the table that will be used to analyze the schema of the incoming CSV table to ensure it matches the expected column definition. If the destination table is not specified, the schema table will be used as the destination table.
-DestTable (string, optional)	This will override the schema table as the destination table where the rows will be inserted.
-BinEndpointId (int, optional)	This is the BIN Endpoint ID that will set the database destination for this operation. If both this and the Alias are unset, the default database will revert to the SERENEDI database.
-BinEndpointAlias (string, optional)	This is the BIN Endpoint Alias that will set the database destination for this operation. If both this and the ID are unset, the default will revert to the SERENEDI database.
-TruncateTable (bool, optional)	Setting this to True will truncate the destination table prior to insertion. This should **never** be set unless the PROCESS_THROTTLE is set to 1 on the base trigger so that this event will run in serial and never in parallel.

sapi-FlatToCSV	This will save the Flat register to a CSV file.
ERRORS	
DT2CSV0010	Datatable is null or empty.
DT2CSV0015	Critical error projecting FLAT to CSV **STR:** exception message
DT2CSV0020	Critical error while setting up CSV
DT2CSV0030	Critical error while creating the rows of the CSV
Parameter	**Value**
-Filename (string, mandatory)	This is the path to the CGIF2 Flat-formatted CSV file to be created.

sapi-FlatFromCSV	This will load the Flat register from a CSV file.
ERRORS	
CSV2DT0010	Filename not Valid
CSV2DT0020	Syntax error parsing CSV
CSV2DT0030	Critical error parsing headers
CSV2DT0040	Critical error parsing rows
CSV2DT0050	Encountered CSV row with different number of columns than expected
CSV2DT0060	Could not establish tree from mappings.
CSV2DT0070	No rows found.
Parameter	**Value**
-Filename (string, mandatory)	This is the path to the CGIF2 Flat-formatted CSV file.

ENVIRONMENT COMMANDS

sapi-ClearRegister	This command can clear individual state machine registers, allowing fine-grained control of the session state machine. This is an alternative to sapi-Reset, which completely refreshes the session state.
ERRORS	
GENR0010	Unknown Register **STR:** register
Parameter	**Value**
-Register (string, mandatory)	Clears one of the specified registers: **HKEY** **SEGPOOL** **MSGLOG** **ACK** **FLAT** **XML**

sapi-EnvEndpointRemove	This command removes an existing database endpoint. The command will fail if there are any BIN items associated with the specified endpoint.
ERRORS	
SENV0080	`Critical error on EndpointRemove` **STR:** `Exception message`
Parameter	**Value**
-BinEndpointID (int, mandatory)	The BIN endpoint to be removed

sapi-EnvEndpointUpsert	This command inserts or updates a BIN endpoint, which is an alias for a predefined database connection. The return value from this command is an **int** of the BIN Endpoint ID that was created, or -1 if this was an update operation.
ERRORS	
SENV0070	`Critical error on EndpointUpsert` **STR:** `exception message`
Parameter	**Value**
-BinEndpointID (int, optional)	This is provided when updating the information of an existing endpoint.
-BinCnnStr (string, optional)	This is the connection string for the database.
-BinDbType (string, optional)	This is the database type. Valid values are: **ORACLE** – Oracle Server **SQLSERVER** – Microsoft SQL Server
-BinEndpointAlias (string, optional)	This is the optional alias that you can assign to the endpoint.

sapi-EnvSFTPSessionUpsert	This will create a new SecureFTP session or update the information in an existing one. The "fingerprint" is not normally set, but will instead be updated to reflect the first SecureFTP session it connects to. If the server and/or fingerprint need to be reset, setting it to **VOID** will return it to its initial state. The return value is an **int** of the new SFTP session created, or -1 if the session was updated.
ERRORS	
SFTP0090	AddSessionFirstConnect Critical Error **STR:** exception message
Parameter	**Value**
-SFTPSessID (integer, optional)	If provided, this will update the values of the SecureFTP session. If not provided, it will update the information associated with this session.
-Hostname	This is either the IP address or hostname of the target SFTP Server.
-Username	This is the username used to log in.
-Password	This is the password used to log in.
-Params	This is a comma-separated list of parameters used for the SFTP session. Parameters represent various state flags that direct the operation of the session. The valid values are: **SCP** – Sessions will open as SCP sessions (SSH Copy) instead of SecureFTP (default). **BINARY** – All transfers will be done using BINARY mode. **ASCII** – All transfers will be done using ASCII mode. **BOTH_MIRROR** – Local and Remote file systems will be mirrored. **REMOTE_MIRROR** – Remote file systems will be mirrored. **REMOVE_FILES** – Files will be deleted during synchronization.
-Fingerprint	If **VOID** is supplied for this parameter, the SecureFTP session fingerprint will be reset until next login.
-PrivateKeyFile	If supplied, this defines a private key file that will be used for authentication.
-PrivateKeyPass	If supplied, this provides a passphrase used to unlock the private key file specified above.

Example:

This example demonstrates setup of a Secure FTP session and its association with a LOCAL_ARCHIVE trigger. Every new file uploaded to the specified remote folder will fire an event once it is mirrored to the local file system. The following SCORE script can be entered into the REPL command line system:

```
md C:\serenedi\pipeline\test_sftp_mirror (Note: for Unix, change the folder to use the Linux pipeline path)
$sftpSessId = (sapi-EnvSFTPSessionUpsert -Hostname <<your sftp server host>> -Username <<user id>> -Password <<password>>)
Write-Host (sapi-EnvTriggerUpsert -TriggerName SFTP_TEST -Script $\Pipeline.ps1 -TriggerType LOCAL_ARCHIVE -InitFolder $\test_sftp_mirror -SourceFolder / -SFTPSessId $sftpSessId -PollInterval 60 -IsEnabled $true -ForceArg3 TEST_SFTP)
```

It will write the ID of the newly created trigger to the console. Given valid SFTP credentials, it will create a trigger that polls the SecureFTP server every 60 seconds for new files, and trigger events when new files are found there.

sapi-EnvSFTPSessionRemove	This will remove an existing SecureFTP session.
ERRORS	
SFTP0080	SessionRemove Critical Error **STR:** exception message
Parameter	**Value**
-SFTPSessID	SFTP Session ID to be removed.

sapi-EnvTriggerRemove	This command will remove an existing trigger. Because of the relational links between the tables, all downstream entries in BIZ_MSG and BIZ_EVENTS, BIN_LOG and BIN_BLOB will need to be removed before the database will allow this trigger to be deleted. Either the ID or the name of the trigger needs to be supplied.
ERRORS	
SENV0060	Critical error on TriggerRemove **STR:** exception message
SENV0065	Invalid Name supplied to TriggerRemove **STR:** trigger name
Parameter	Value
-TriggerID (integer, optional)	This is the trigger to be removed from the BIZ_TRIGGER table.
-TriggerName (string, optional)	This is the name of the trigger to be removed.

sapi-EnvTriggerUpsert	This command will enable scripts to add new triggers or update existing triggers in the Event system. The Trigger system is explained further in the "Events" chapter. The return value is the **int** value BizTriggerId of the new trigger if inserting, or -1 if this is an update operation.
Parameter	**Value**
-TriggerID (integer, optional)	If you're updating an existing trigger, this is the Trigger ID to modify. If this is not provided, then the action will be treated as a brand-new trigger.
-TriggerName (string, optional)	This is the optional name of the trigger. It can be used in lieu of the Trigger ID when making updates.
-Script (string, optional)	This is the path to execute the PowerShell Core script that will be run when the trigger is fired. For new triggers, this is mandatory.
-TriggerType (string, optional)	This establishes the firing criteria for the trigger. The valid values are: **LOCAL_UPLOAD** **LOCAL_ARCHIVE** **SQL**
-InitFolder (string, optional)	For Upload trigger types, this is the initial folder where files must be placed to fire the trigger. The act of successfully moving the file from the Init folder to the source folder is the primary firing criterion for Upload triggers. For Archive trigger types, the primary firing criterion is finding a new file in the Init folder that was not previously used to fire an event.
-SourceFolder (string, optional)	For triggers not bound to an SFTP session, this specifies the destination folder for Upload triggers. For LOCAL_ARCHIVE triggers bound to SFTP sessions, this specifies the remote folder for the SFTP session.
-SFTPSessID (integer, optional)	This is the SecureFTP session that establishes a *mirror* with a remote file archive. Every Poll Interval, the local mirror will be refreshed by this SecureFTP session as described in the **Event system**.

-FireLogic (string, optional)	This establishes various filters that set conditions for firing events based on groupings of files and/or month/day/time. More information is available in the "**Event System**" section.
-SFTPPollDt (Date Time, optional)	This is the last time this trigger's SecureFTP session was scanned.
-LastFireDate (Date Time, optional)	This is the last time the trigger was fired.
-PollInterval (integer, optional)	This is the number of seconds the Event system will wait between checks for this trigger's firing criteria.
-IsEnabled (Boolean, optional)	True will enable the trigger; False will disable it.
-MaxProcess (integer, optional)	This is the maximum number of simultaneous executions allowed for a trigger. Setting this to 1 will limit the trigger to **serial** execution and prevent all parallel execution.
-ForceArg1 (string, optional)	When provided, this will pass a fixed value to the trigger script for Archive and SQL type triggers. It cannot be used for Upload trigger types.
-ForceArg2 (string, optional)	When provided, this will pass a fixed value to all trigger types.
-ForceArg3 (string, optional)	When provided, this will pass a fixed value to all trigger types.
-ForceArg4 (string, optional)	When provided, this will pass a fixed value to all trigger types except the Immediate Event type, which uses Event Argument 4 to provide the path to a PowerShell Core script.
ERRORS	
SENV0050	`Critical error on TriggerUpsert` **STR:** `exception message`

sapi-FetchVar	This command pulls data from the SERENEDI session object to allow you to see the internal state. The return value from this command is determined by the parameter passed to it.
ERRORS	
MISC0010	`Critical error fetching Variable <<Value>>`
Parameter	**Value**
-Value (string, mandatory)	**SEG** – This returns a single-character Segment Separator.
	ELE – This returns a single-character Element Separator.

	SUBELE – This returns a single-character composite Element Separator.
	ELEREPEAT – This returns a single-character Element Repeat separator.
	SEG_CT – This returns the number of loaded segments in the SegPool register.
	ACK_CT – This returns the number of loaded segments in the Acknowledgment register.
	SPEC_CD – This returns the Specification Code for the active specification. See "Appendix: Specification Codes" for the list of return values.
	SPEC_NM – This returns the short specification name for the active specification. See "Appendix: Specification Codes" for the list of return values.
	CRIT_ERR – This returns Boolean True or False depending on the critical error status of the SERENEDI session state.
	FLAT_COL_CT – This returns 0 if no Flat register is loaded, or the column count of the Flat register if it is.
	FLAT_ROW_CT – This returns 0 if no Flat register is loaded, or the row count of the Flat register if it is.
	FLAT_DT – This returns a C# DataTable object representing the active Flat register.
	"HKEY_DT" – If the HKey register is set, this will return a Dictionary<string, DataTable> object that holds the HKey rendered as a series of datatables. Each Key Value Pair within the Dictionary object is Keyed with 'HDB_ISA' for the first table, 'HDB_GSHDR' for the second table, and so on for all of the loops present within the HKey. The Data Tables follow the same specifications as the HDB data system defined earlier in the documentation.
	HKEY_XML – This returns an XDocument object representing the XML.
	HKEY_XML_LEN – This returns a 0 if the XML register is not loaded; otherwise, it will return the string length of the XML.

	HKEY_INFO – This returns two bar-separated numbers reflecting the Loop count and Element count of the loaded HKey register. For example: 480|1843
	MSG_CT – This returns the number of messages in the MsgLog register.
	TREE_NM – This returns the short specification of the loaded Tree, or writes Unloaded if the tree is not loaded.
	MSG_HTML – This returns an HTML dump of all messages.
	MSG_XML – This returns the message log in XML format.
	SEG_TEXTBLOCK – This emits the text of the SegPool, along with the established text division characters, to the console output.
	LOOP_LIST – This outputs a space-delimited list of all the loops (loaded with data or not) associated with the active loaded specification.
	DEFAULT_ICN **DEFAULT_GCN** **DEFAULT_TCN** These return the default values of the ICN/GCN/TCN envelope control numbers.

Example

Dumping error messages to an HTML file:

```
sapi-FetchVar -Value “MSG_HTML” | Out-File “C:\serenedi\msg.html”
```

Dumping error messages to an XML file:

```
sapi-FetchVar -Value “MSG_XML” | Out-File “C:\serenedi\msg.xml”
```

sapi-InitializeSession	This command sets up a new instance of the SERENEDI engine, which is normally handled for you by the SERENEDI runtime environment. If you'd like to use an external debugger and step through the code, however, you need to handle this yourself from within a PowerShell Core scripting environment such as Visual Code.
	
	Normally, it's the third in a set of commands that initialize the environment:
	
	`Set-Location C:\serenedi\bin`
	`Import-Module -Name (Resolve-Path 'SERENEDI2.dll')`
	`sapi-InitializeSession -BaseDir 'C:\serenedi'`

ERRORS	
`Emergency Error Log in Base Directory`	`InitializeSession Spawn Error:` **`exception message`**
Parameter	**Value**
`-BaseDir (string, mandatory)`	This tells SERENEDI the base SERENEDI folder (such as C:\serenedi or /opt/serenedi) so it is able to locate other critical resources.
`-BizEventId (int, optional)`	When provided, the SERENEDI session's global variables will be preset to the values associated with an already-fired event. This can be useful for determining exactly what happened during that event that caused an error.

sapi-Reset	This will completely reset the active session state. If you need to process two EDI files within a script session, it's best to reset the session between files so that various internal registers specific to that transaction are cleared.

INTEGRITY COMMANDS

sapi-AddIntegrityRule	This command adds a custom rule to the integrity rules engine. The RuleCode parameter must be a valid REPCode Boolean expression. The rule will add the provided message to the message log when the REPCode expression evaluates to True during a decode operation.
ERRORS	

RE0010	Rule Engine Add Rule Failure **STR:** exception message
RE0020	Rule Engine Loop Unknown **STR:** loop name **N1:** rule order
TOKN0010	Invalid Map in Expression **STR:** invalid map
TOKN0020	Failure to Tokenize **STR:** REPCode text
TOKN0030	Failure to Parse Tokens **STR:** REPCode text
Parameter	**Value**
-SpecCd (string, mandatory)	This value represents all the two-digit specification codes that are linked for this rule.
-LoopNm (string, mandatory)	This is the loop short name (like L2300) that links to this rule. Every time this loop is encountered during a decode, this rule is executed.
-RuleOrder (int, mandatory)	This is the order in which the rule is executed. Normally, it starts at more than 10000 so it does not conflict with baseline integrity rules.
-RuleCode (string, mandatory)	This is the REPCode of the rule itself.
-Message (string, mandatory)	This is the message added to the message log when this rule is triggered during a decode.
-ShowXMLTokens (flag, optional)	When this flag is given, an XML file showing the node composition of the supplied REPCode is returned as a string value.

## sapi-CheckIntegrity	This command requires that both the SegPool and the HKey register be loaded, and will conduct a deep integrity check of the file. Currently, the 834, 835, and 837 I & 837 P specifications are the only supported transactions for the Deep Integrity analysis engine. This command has no effect if run on other transactions. Over 300 different errors are supported for these specifications. Errors are identified by the Loop Short Name and the Rule Order. These can be individually disabled with the DisableIntegrityRule command.
ERRORS	

INTEG0010	SegPool must be loaded and then decoded prior to an Integrity operation.

sapi-DisableIntegrityRule	
ERRORS	
RE0020	Rule Engine Loop Unknown
RE0030	Rule Engine Order Unknown
RE0040	Rule Engine Disable Failure
Parameter	**Value**
-SpecCd (string, mandatory)	This is the specification code of the rule to disable.
-LoopNm (string, mandatory)	This is the Short Loop Name of the rule to disable.
-RuleOrder (int, mandatory)	The is the Rule Order of the rule to disable.

MSGLOG COMMANDS

sapi-AddMsg	This command adds a custom message to the MsgLog register.
Parameter	**Value**
-Origin (string, optional)	The origin is a short string that identifies the source of the message. It will default to USER if not supplied.
-Message (string, mandatory)	This is the required primary message.
-StringData (string, optional)	This is additional string data that gives context to the primary message.
-IntData1 (integer, optional)	This is Integer Data 1 to give additional information to the message.
-IntData2 (integer, optional)	This is Integer Data 2 to give additional information to the message.

sapi-GetMsg	This fetches a specific message from the MsgLog Session State register. Use the command sapi-FetchVar -Value 'ERR_CT' to obtain the total number of messages available. The return value is a string of a single message. Messages consists of five fields, separated by the pipe character |. The format of the message is: Origin | Message | String Data 1 | Integer Data 2 | Integer Data 3
ERRORS	
MSG0010	Invalid Message ID
Parameter	Value
-MessageID (int, mandatory)	This is the index of the message to fetch.

sapi-MsgLogToFile	This command allows dumping of the current message log into a CSV file in the file system in HTML format.
ERRORS	
MSG0020	Unable to write file **STR:** exception message
Parameter	Value
-Filename (string, mandatory)	This is the file to write the HTML-formatted message log.

sapi-MsgLogToHTML	This cmdlet will return the HTML as a string. The return value is an HTML string of the message log.
ERRORS	
MSG0030	Unable to render msglog **STR:** exception message

REGISTER COMMANDS

sapi-AckFromFile	This will load the Acknowledgment register from a 999 file in the file system.
ERRORS	
ACK0010	Failure reading stream **STR:** exception message

Parameter	Value
-Filename (string, mandatory)	This is the file path to a valid 999 transaction.

sapi-AckFromHKey	This will project the HKey to the Acknowledgment register. It is predicated on the HKey being loaded with a valid 999 Acknowledgment transaction. This command inherits the errors from the SegPoolFromHKey command.

sapi-AckFromSegPool	This transfers the loaded SegPool register to the Acknowledgment register and forces the session tree variable to the 999 specification.

sapi-AckToFile	This will save the Acknowledgment register to a 999 file in the file system.
ERRORS	
ACK0020	Error generating ACK to File **STR:** exception message
Parameter	**Value**
-Filename (string, mandatory)	This is the file path to a valid 999 transaction to be created.
-Formatting (string, optional)	This four-character string enables you to override the encoder to use different text division characters: Position 1: Element Separator Default: ***** Position 2: Segment Separator Default: **~** Position 3: Composite Element Separator Default: **:** Position 4: Repeating Element Default: **^**
-bolCR (bool, optional)	When true, inserts a Carriage Return (Char 13) after each segment. Default: True.
-bolLF (bool, optional)	When true, inserts a Line Feed (Char 10) after each segment. Default: True.

sapi-AckToHKey	This will project the Acknowledgment register to the HKey register. It inherits the errors from the SegPoolToHKey command.

sapi-AckToSegPool	This transfers the active Acknowledgment register to the SegPool register.

sapi-FlatFromHKey	This command executes a translation from the Flat register to the HKey register.
ERRORS	
H2F0010	Critical error during ScanRows **STR:** exception message
H2F0020	Critical error during ScanRows **STR:** exception message
H2F0030	Critical error during Copy Previous Row **STR:** exception message
H2F0040	Critical error during Populate RDR **STR:** exception message
H2F0050	Critical error during SpawnDataRow **STR:** exception message
H2F0060	Tree not loaded.

sapi-FlatToHKey	This command executes a translation from the HKey register to the Flat register.
ERRORS	
F2H0010	Flat Decode Critical Failure **STR:** exception message
F2H0020	Flat Decode Critical Failure – Unparsable Loop **STR:** name of last parsed loop **N1:** number of unparsed loops remaining
F2H0040	Flat Unsupported Data Type **STR: column name** Unsupported Type: **type name** **N1:** column index
F2H0050	Column cannot be parsed **STR:** column name **N1:** column index

F2H0060	Inherited Iteration mapping is referencing a loop iteration value not present in the Single Iteration parent loop maps **STR:** mapping name
F2H0065	Value Iteration mapping is referencing a loop iteration value not present in the Single Iteration parent loop maps **STR:** mapping name
F2H0070	Critical Flat Data Conversion Error **STR:** column name
F2H0080	Inherited Value mapping is referencing a value not present in the Value Qualified parent loop maps. **STR:** column name
F2H0090	Flat register is unloaded.
F2H0100	Last column must be NEWROW to be a valid Flat.
F2H0110	Flat Decode Critical Error – Could not parse data stream **STR:** last parsed short loop name

sapi-GenerateAck	This command will generate a generic 999 Acknowledgment transaction based on the currently loaded and processed SegPool, HKey, and MsgLog registers. If the most recent SegPool to HKey translation was successful, it will generate a Transaction Accepted 999 Acknowledgment. If the translation failed, it will generate a File Rejected Acknowledgment and specify the segment at which translation failed.
ERRORS	
ACK0010	There was an exception while attempting to generate the 999. **STR:** exception message
ACK0020	GenerateAck error – no SegPool loaded
ACK0030	GenerateAck error – SegPool not decoded

sapi-ParseAck	This command will parse a loaded Acknowledgement and generate a series of messages based on the contents. When a SegPool is loaded with the transaction this Acknowledgement was generated against, these messages can make it easier to understand why a particular transaction was rejected. The messages are defined in the HIPAA Implementation Guide for 999 specification.
ERRORS	
ACK0040	Parse Acknowledgment – no Acknowledgment loaded.

sapi-SegPoolFromFile	This command loads an EDI transaction composed of elements and segments into the SegPool register. Once it is loaded successfully, the active specification is set based on the contents of the file.
Parameter	**Value**
-Filename (string, mandatory)	This is the path of the file system EDI file.
ERRORS	
SEG0010	There was a low-level critical error loading the SegPool
SEG0020	Filename not valid
SEG0030	ConsumeTextStream encountered a critical error.
SEG0040	Insufficient segments were found on the incoming file.
SEG0050	Unknown Exception **STR:** exception message
SEG0060	An ISA/IEA envelope in the stream is invalid because it has non-unique partitioning characters **STR:** ELESEP/SUBELE/ELERPT/SEGSEP **partitioning characters**
SEG0070	A critical error was encountered while parsing the incoming SegPool stream. **STR:** exception message
SEG0080	There was a critical error generating the text stream for the SegPool. **STR:** exception message
SEG0085	FetchSpecificSegment is being accessed with an incorrect SegSlice index.
SEG0090	There was a critical error when fetching the specified segment from the SegPool.

	STR: exception message **N1:** segment index
SEG0100	There was a critical error when fetching the specified segment from the SegPool **STR:** exception message **N1:** segment index
SEG0110	There was a critical error encountered when adding a new segment to the SegPool. **STR:** exception message
SEG0120	There was a critical error encountered while closing the SegPool. **STR:** exception message

sapi-SegPoolFromHKey	This will translate the HKey register to the SegPool register.
ERRORS	
H2SEG0010	SegPool_from_HKey Critical Error **STR:** exception message **N1:** segment count
H2SEG0020	Tree is undefined – encoding failed
H2SEG0030	HKey is unloaded – encoding failed
H2SEG0040	Critical Error during SegPool Encode **N1:** segment count

sapi-SegPoolToFile	This will generate a new file system object from the loaded SegPool register.
ERRORS	
SEG2F0010	No SegPool loaded
SEG2F0020	There was a critical error generating the text stream for the SegPool. **STR:** exception message
Parameter	**Value**
-Filename (string, mandatory)	This is the path of the file system EDI file to be created.
-Formatting (string, optional)	This four-character string enables you to override the SegPool encoder to use different text-division characters: Position 1: Element Separator Default: ***** Position 2: Segment Separator Default: **~**

	Position 3: Composite Element Separator Default: **:** Position 4: Repeating Element Default: **^**
-bolCR (bool, optional)	When true, inserts a carriage return (Char 13) after each segment. Default: True.
-bolLF (bool, optional)	When true, inserts a line feed (Char 10) after each segment. Default: True.

sapi-SegPoolToHKey	This will translate the SegPool to the HKey register. Note: all error messages listed below will store the segment index of the time of the error in the **N1** field.
ERRORS	
H2SEG0010	Invalid Date Length **STR:** element
H2SEG0020	Invalid Time Length **STR:** element
H2SEG0040	Element Requirement conditions not met
H2SEG0050	Invalid Qualifiers in DTP segment
H2SEG0060	Date stamp is not in a valid character format
H2SEG0070	DateTime stamp is not in a valid character format
H2SEG0080	Time stamp is not in a valid character format
H2SEG0090	RD8 date stamp is not in a valid 17 character format
H2SEG0100	Data present on an element marked as unused **STR:** element index
H2SEG0110	Data is below the minimum length for this element **STR:** element
H2SEG0120	Data exceeds the maximum length for this element **STR:** element
H2SEG0130	Data not present on an element marked as required **STR:** element index
H2SEG0140	Repeating element exceeds the allowed number of iterations **STR:** element index
H2SEG0150	Specified repeating element is not found in validating code list **STR:** element index

H2SEG0160	Specified repeating composite element is not found in validating code list **STR:** element index
H2SEG0170	Specified value-defined element is not valid
H2SEG0180	Specified qualifier element is not valid
H2SEG0190	Data present on a composite element marked as unused **STR:** element index
H2SEG0200	Specified composite value-defined element is not valid **STR:** element index
H2SEG0210	Specified composite value-defined element is not valid **STR:** element index
H2SEG0220	Segment present without elements
H2SEG0230	Required segment is missing **STR:** Short Loop Name : Segment Name
H2SEG0240	Segment exceeds maximum iterations **STR:** Short Loop Name : Segment Name
H2SEG0250	Required loop is not present **STR:** Short Loop Name
H2SEG0260	Loop has too many iterations **STR:** Short Loop Name
H2SEG0270	SegPool is not loaded.
H2SEG0280	Unable to determine specification – check the ISA/GS/ST/BHT segments for formatting issues
H2SEG0290	Critical error during decoding on setup **STR:** exception message
H2SEG0300	Premature End of File **STR:** invalid segment
H2SEG0310	Invalid HL Child Indicator **STR:** Inbound HL04 **hl04** Expected HL04 **expected**
H2SEG0320	Critical error during decoding during parse **STR:** exception message
H2SEG0330	Critical error during decoding
H2SEG0340	Could not decode file **STR:** invalid segment
Parameter	**Value**

-EnableCodeSetChecks (flag, optional)	This option will load the code sets that are supported by SERENEDI and raise error messages when invalid codes are used within segments.

sapi-SegPoolToHTML	This will create an HTML view of the SegPool along with any messages loaded highlighted in red. The return value from this cmdlet is a **string** of the SegPool in HTML form. Additionally, any messages in the MsgLog will be displayed.
ERRORS	
S2HTML0010	Critical error creating HTML **STR:** exception message

sapi-SetFlat	This command allows you to manually set the internal Flat register from a DataTable formatted with valid CGIF2 maps. The DataTable must be a fully compliant Flat DataTable, including an ending NEWROW integer column, a specification tag in the first map, and correct maps for all columns. The sapi-FetchVar -Register 'FLAT_DT' command is the inverse of this command and can be used to retrieve the Flat DataTable.
Parameter	**Value**
-DT (DataTable, mandatory)	DataTable to set. This is for advanced users who need to bypass the normal methods of loading the Flat register.

SFTP COMMANDS

sapi-GetSFTPDirectory	Provided a predefined SecureFTP Session ID and a remote folder, this will return a string array of the directory contents. Each string corresponds to either a file or a directory. If it is a directory, the value will be the directory name and ending in a | character. If the entry is a file, the value will be the file name, a | character, and the file size in bytes.
Parameter	**Value**
-SFTPSessID (integer, mandatory)	This is the unique ID of the predefined SecureFTP session.
-RemoteFolder (string, mandatory)	This is the remote folder to be retrieved.
ERRORS	

SFTP0140	Directory Failure **STR:** remote directory

sapi-GetSFTPFile	This will fetch a remote file from a predefined SecureFTP session to the local file system.
Parameter	**Value**
-SFTPSessID (int, mandatory)	This is the unique ID of the predefined SecureFTP session.
-LocalFile (string, mandatory)	This is the local file to be created.
-RemoteFile (string, mandatory)	This is the remote file to fetch.
ERRORS & MESSAGES	
SFTP0030	File Downloaded **STR:** filename
SFTP0040	SFTP Download Failure **STR:** filename

sapi-PutSFTPFile	This will push a local file to a remote SecureFTP directory.
Parameter	**Value**
-SFTPSessID (integer, mandatory)	This is the unique ID of the predefined SecureFTP session.
-LocalFile (string, mandatory)	This is the local file to be pushed to the remote file system.
-RemoteFile (string, mandatory)	This specifies the name of the file as it will exist on the remote file system.
ERRORS & MESSAGES	
SFTP0120	File Uploaded **STR:** filename
SFTP0130	SFTP Upload Failure **STR:** filename

sapi-SFTPMirror	This command mirrors a remote file system with the local file system. A SecureFTP option is associated with the session that determines the behavior of the mirror operation. These are set via the sapi-EnvSFTPOptionUpsert / Remove commands. The available options are: **BOTH_MIRROR**: New files in the local folder will be uploaded to the remote server, and new remote server files will be downloaded to the local folder. **REMOTE_MIRROR**: New files in the local folder will be uploaded to the remote server. **FILE_MOVE**: After a successful file transfer, the source file is removed. The default behavior is to locally mirror: new server files will be downloaded to the local folder.
Parameter	**Value**
-SFTPSessID (integer, mandatory)	This is the unique ID of the predefined SecureFTP session.
-RemoteFolder (string, mandatory)	This is the remote folder on the SecureFTP server to mirror.
-LocalFolder (string, mandatory)	This is the local file system folder that will mirror the remote folder.

SQL COMMANDS

sapi-ExecSQL	This executes arbitrary SQL in the specified database.
Parameter	**Value**
-SQL (string, mandatory)	SQL to be executed.
-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.
-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
ERRORS & MESSAGES	
ORCL0010	SQL Error: **<<sql executed>>** **STR:** exception message
SQL0020	SQL Error: **<<sql executed>>** **STR:** exception message

sapi-FetchDTFromDB	This command will return a DataTable based on a passed SQL string. The database connection will default to the SERENEDI database unless overridden via the BIN Endpoint ID or BIN Endpoint Alias. This returns a DataTable of the SQL results.
Parameter	**Value**
-SQL (string, mandatory)	This command will be used to execute a database SQL command and then return the result set as a .NET DataTable object.
-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.
-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
-NoTypeCheck (Boolean, optional)	If set to True, this command will return a *raw* DataTable from an Oracle database without first converting certain columns to **int** data types. This has no effect for SQL Server database connections.
ERRORS & MESSAGES	
SQL0100	`FetchDTFromDB critical error` **STR:** exception message
ORSQL0040	`SQL Error` **STR:** exception message
ORSQL0045	`Error retrieving FLAT from Oracle` **STR:** exception message

sapi-FetchDTFromDB1Row	This command will return a single-row DataTable based on a passed SQL string. The database connection will default to the SERENEDI database unless overridden via the BIN Endpoint ID or BIN Endpoint Alias.
Parameter	**Value**
-SQL (string, mandatory)	This command will be used to execute a database SQL command and then return the result set as a .NET DataTable object. Only the first row will be returned.
-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.

-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
ERRORS & MESSAGES	
SQL0100	FetchDTFromDB critical error **STR:** exception message
ORSQL0040	SQL Error **STR:** exception message

sapi-FetchScalar	This command will return a single-row DataTable based on a passed SQL string. The database connection will default to the SERENEDI database unless overridden via the BIN Endpoint ID or BIN Endpoint Alias.
Parameter	**Value**
-SQL (string, mandatory)	This command will be used to execute a database SQL command and then return the result set as a .NET DataTable object. Only the first row will be returned.
-BinEndpointId (int, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint ID.
-BinEndpointAlias (string, optional)	When given, this will override the default database connection and use the one defined in the BIN system using the BIN Endpoint Alias.
ERRORS & MESSAGES	
ORCL0020	SQL Error: **<<sql executed>>** **STR:** exception message
SQLS0010	Error while FetchingScalar: **<<sql executed>>** **STR:** exception message

XML COMMANDS

sapi-HKeyFromXml	This will load the HKey register from the XML register.
ERROR	
X2H0010	Critical Error during XML Decode **STR:** last XML **N1:** # of XML loops remaining
X2H0020	HKey not loaded – cannot encode

X2H0030	Critical Error during XML Iterate Loop **STR:** last XML **N1:** # of XML loops remaining

sapi-HKeyToXml	This will save the HKey register to the XML register.
ERROR	
H2X0010	Critical Error during XML Encoding **STR:** exception message **N1:** segpool segment ID
H2X0020	HKey not loaded – cannot encode
H2X0030	Critical error during XML Encode **STR:** exception message

sapi-SetXML	This command allows you to manually set the internal XML register from the provided XML text.
ERROR	
MISC0010	Invalid XML **STR:** exception message
Parameter	**Value**
-XML (string, mandatory)	String of the XML to load. This must be formatted as a valid CGIF2 XML object.

sapi-XmlFromFile	This will load the XML register from a file.
Parameter	**Value**
-Filename (string, mandatory)	This is the path of the CGIF2-formatted XML file to be loaded.
ERROR	
XMFF0010	Filename not Valid **STR:** filename
XMFF0020	Critical error loading XML **STR:** exception message

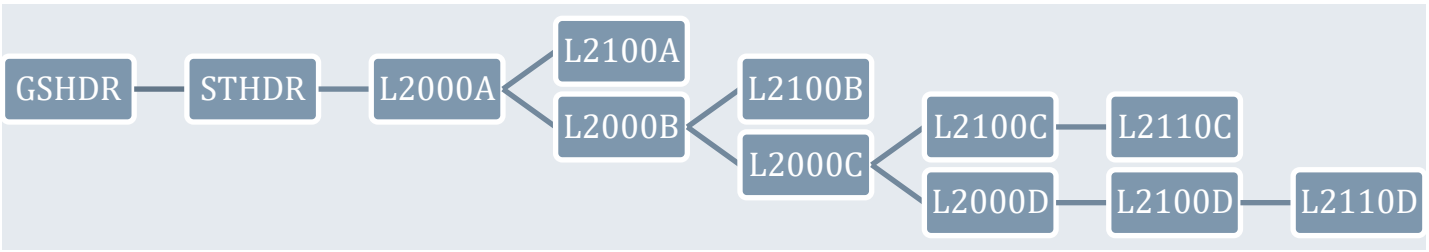
sapi-XmlToFile	This will save the XML register to a file.
Parameter	**Value**
-Filename (string, mandatory)	This is the path to the CGIF2-formatted XML to be created.
ERROR	
XMFF0030	Critical error saving XML **STR:** exception message

Appendix B: Specification Hierarchy Structures

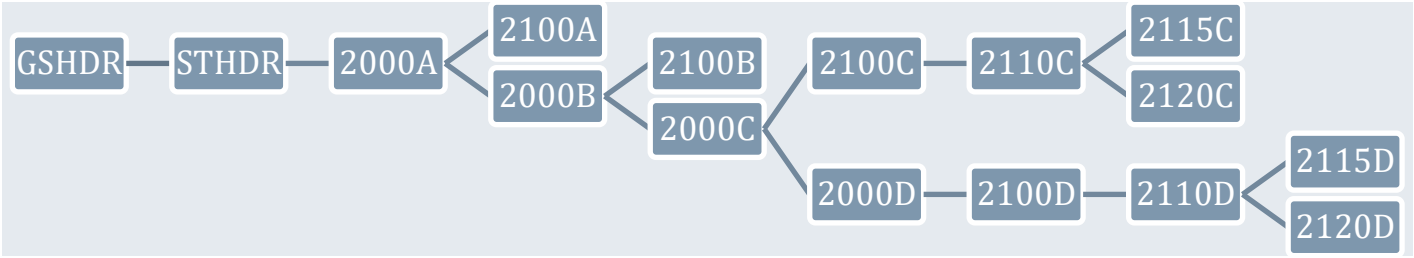
These diagrams show the hierarchical relationship between the loops within a transaction. This is important for knowing the parent/child loop relationships present within the HDB tables so that you know which table a parent ID is pointing to in a child table.

Any loop that has a name ending in X or Y is a *cutout* loop. These loops are not defined in the HIPAA Implementation Guides – instead, they are a SERENEDI convention in which a single segment is pulled from the parent loop because they are defined as having unlimited repeats. Keeping this information isolated in its own dedicated loop makes the data easier to access for both encoding and decoding.

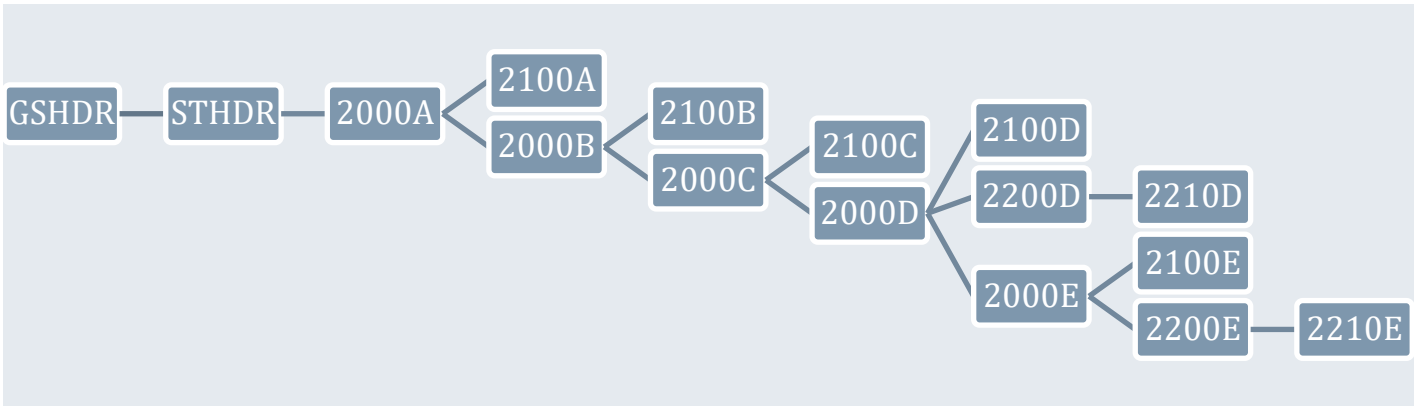
5010_270 / M0 Health Care Eligibility Benefit Inquiry



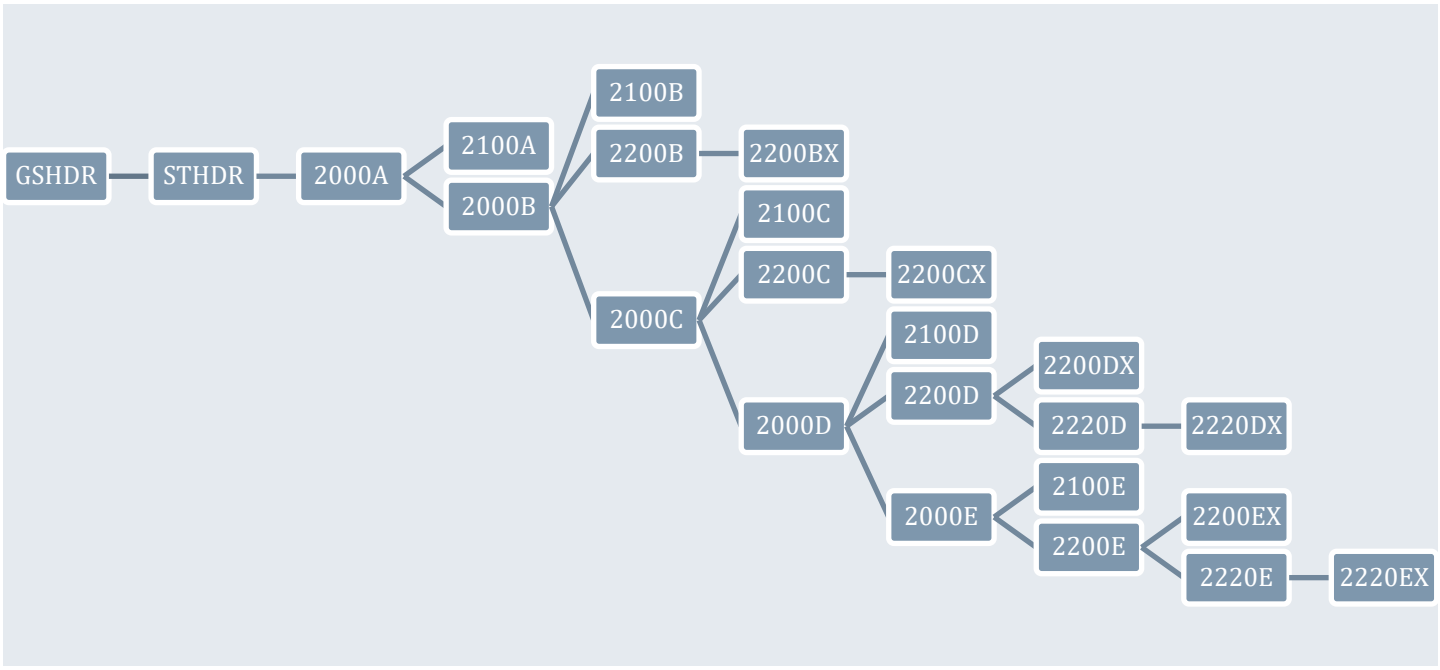
5010_271 / N0 Health Care Eligibility Benefit Response



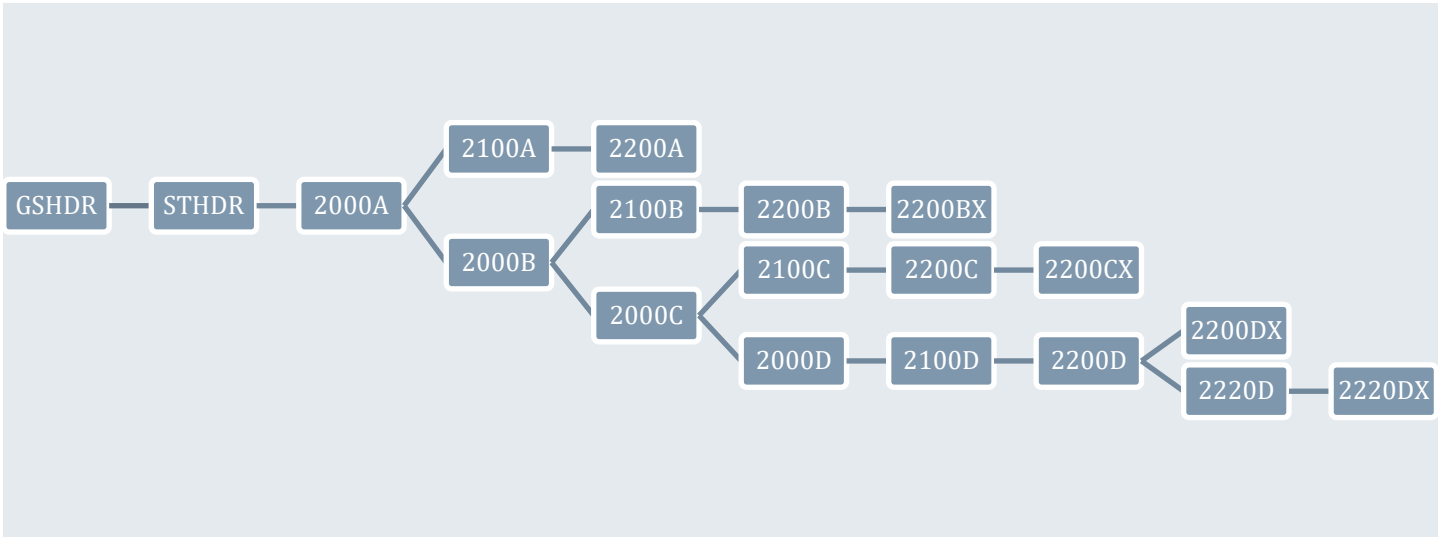
5010_276 / O0 Health Care Claim Status Request


GSHDR
STHDR
2000A
2100A
2000B
2100B
2000C
2100C
2000D
2100D
2200D
2210D
2000E
2100E
2200E
2210E

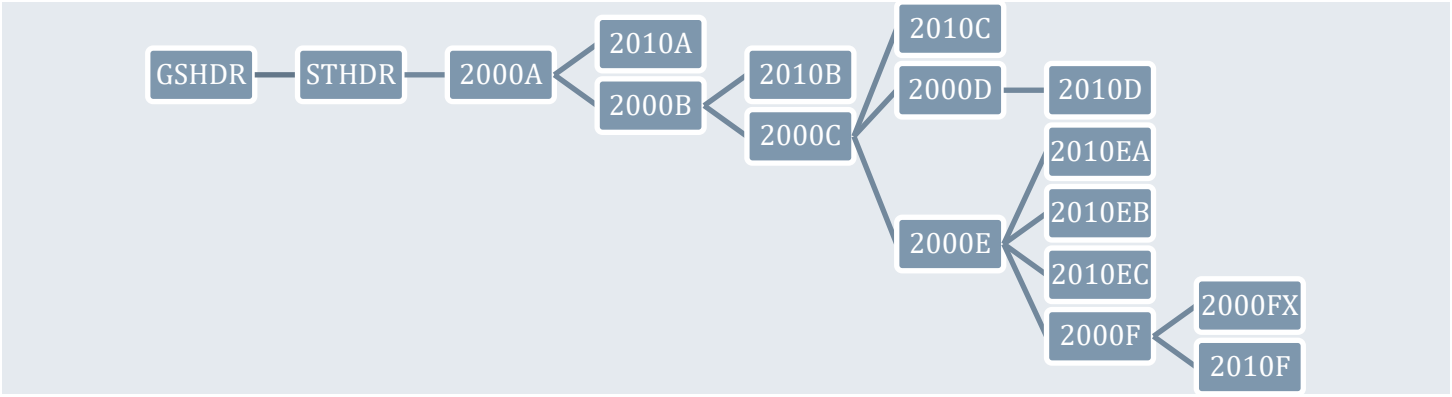
5010_277 / P0 Health Care Claim Status Response



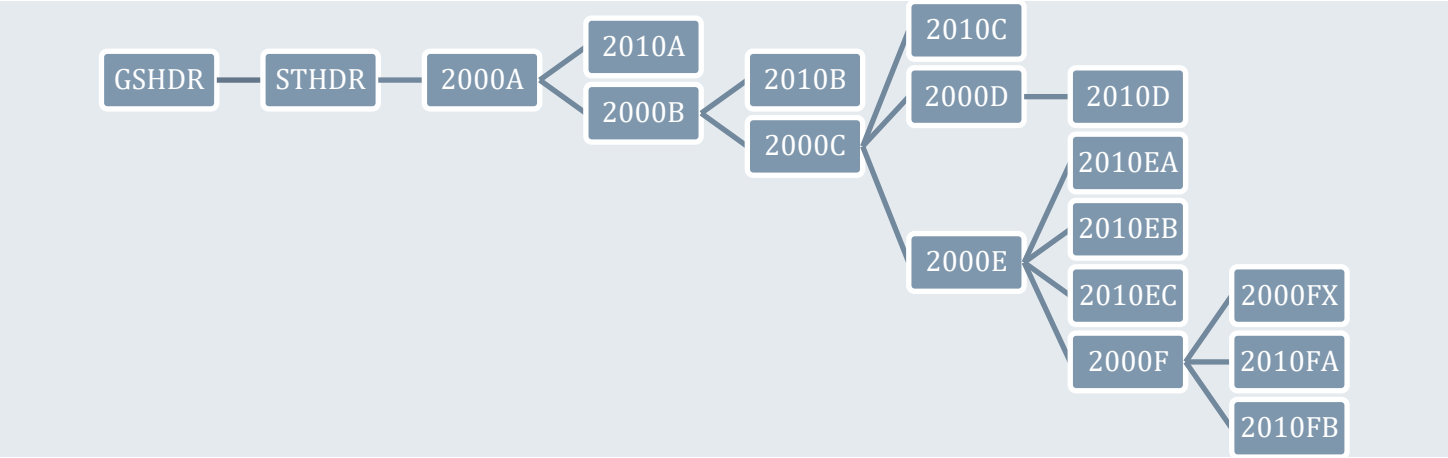
5010_277CA / P5 Health Care Claim Acknowledgment



5010_278_REQ / Q0 Health Care Services Review – Request for Review

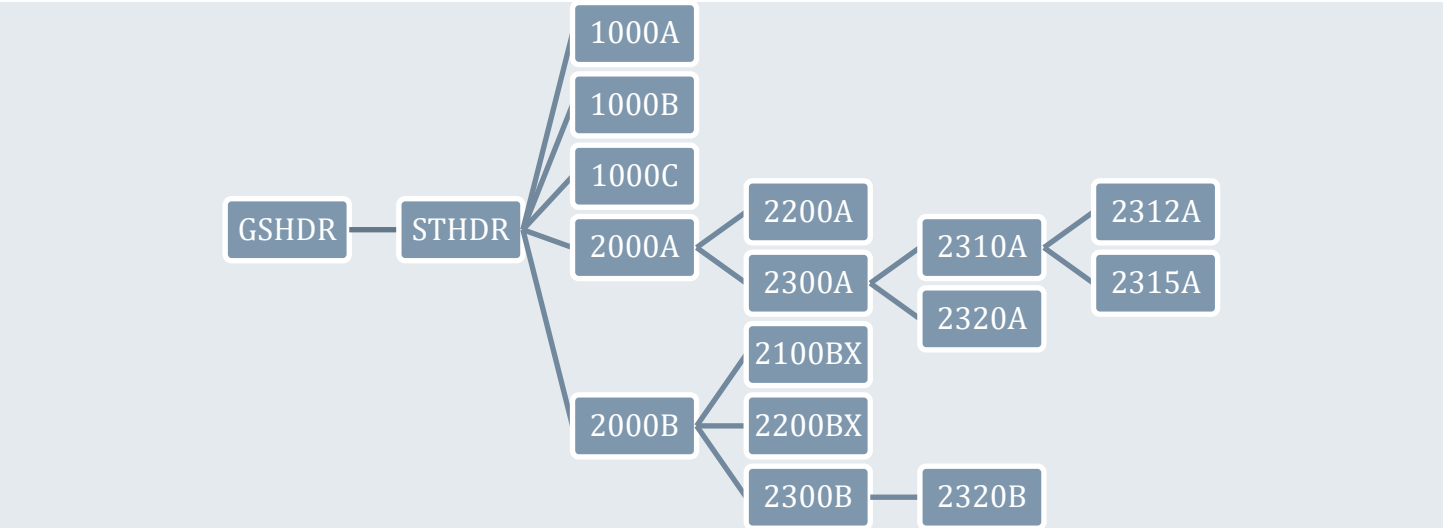


5010_278_RESP / R0 Health Care Services Review – Response

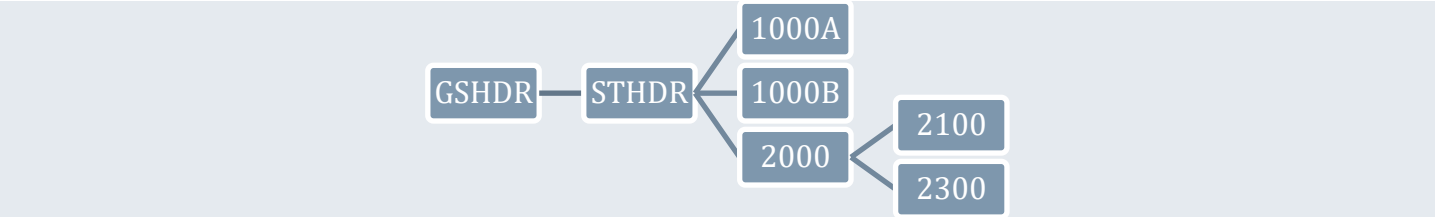


5010_820 / S0

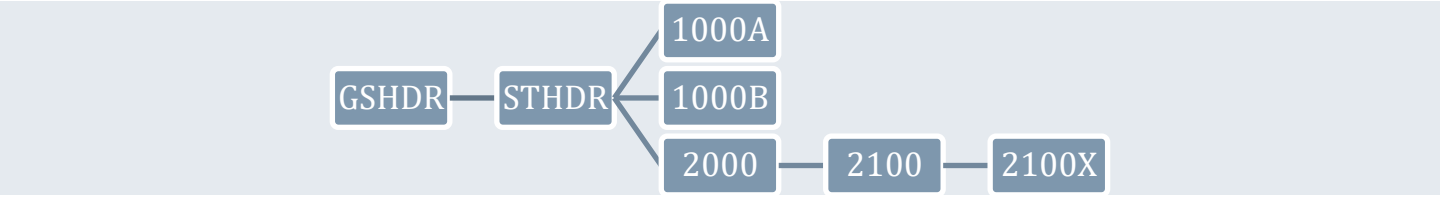
Payroll Deducted and Other Group Premium Payment for Insurance Products



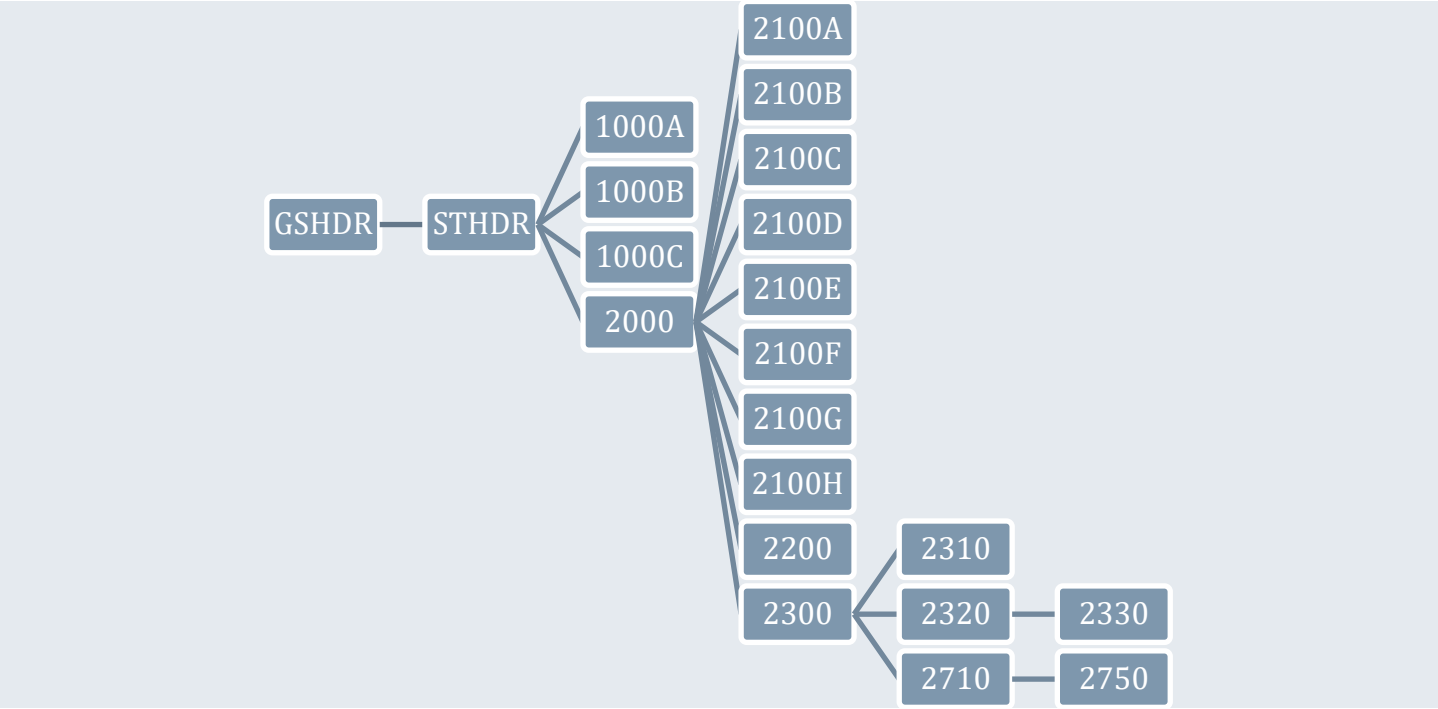
5010_820X / S5 Health Insurance Exchange Related Payments



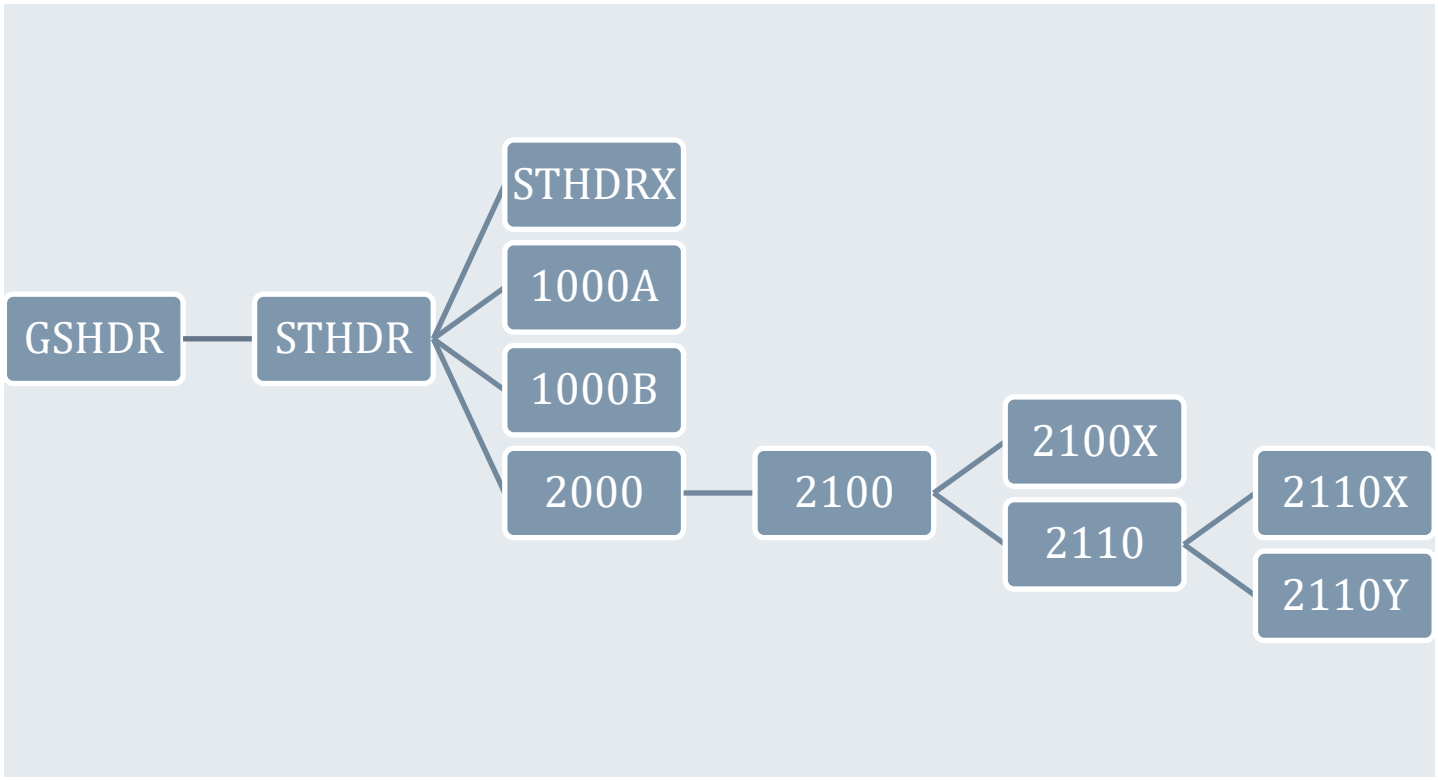
5010_824 / P7 Application Reporting for Insurance



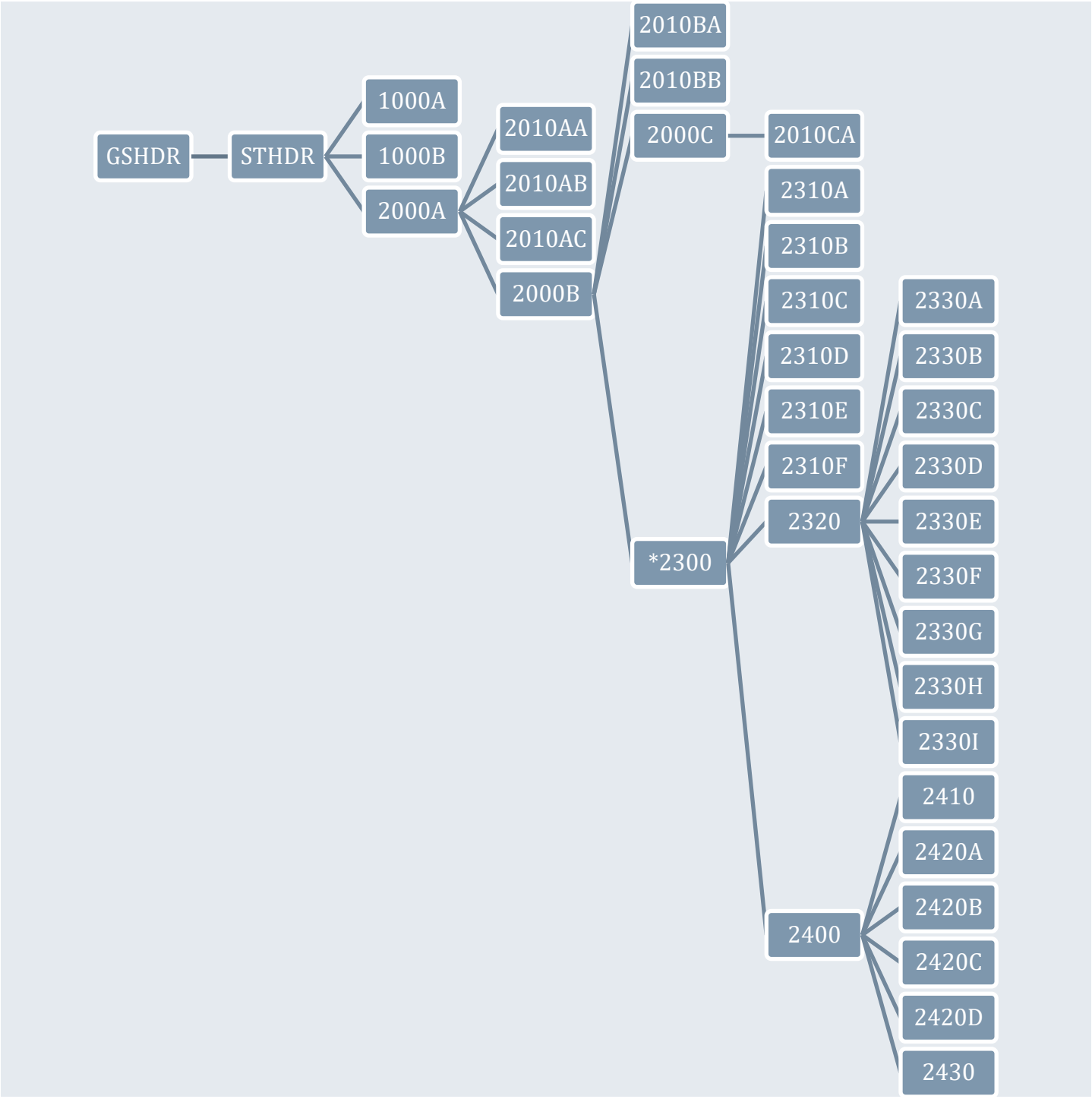
5010_834 / T0 Benefit Enrollment and Maintenance



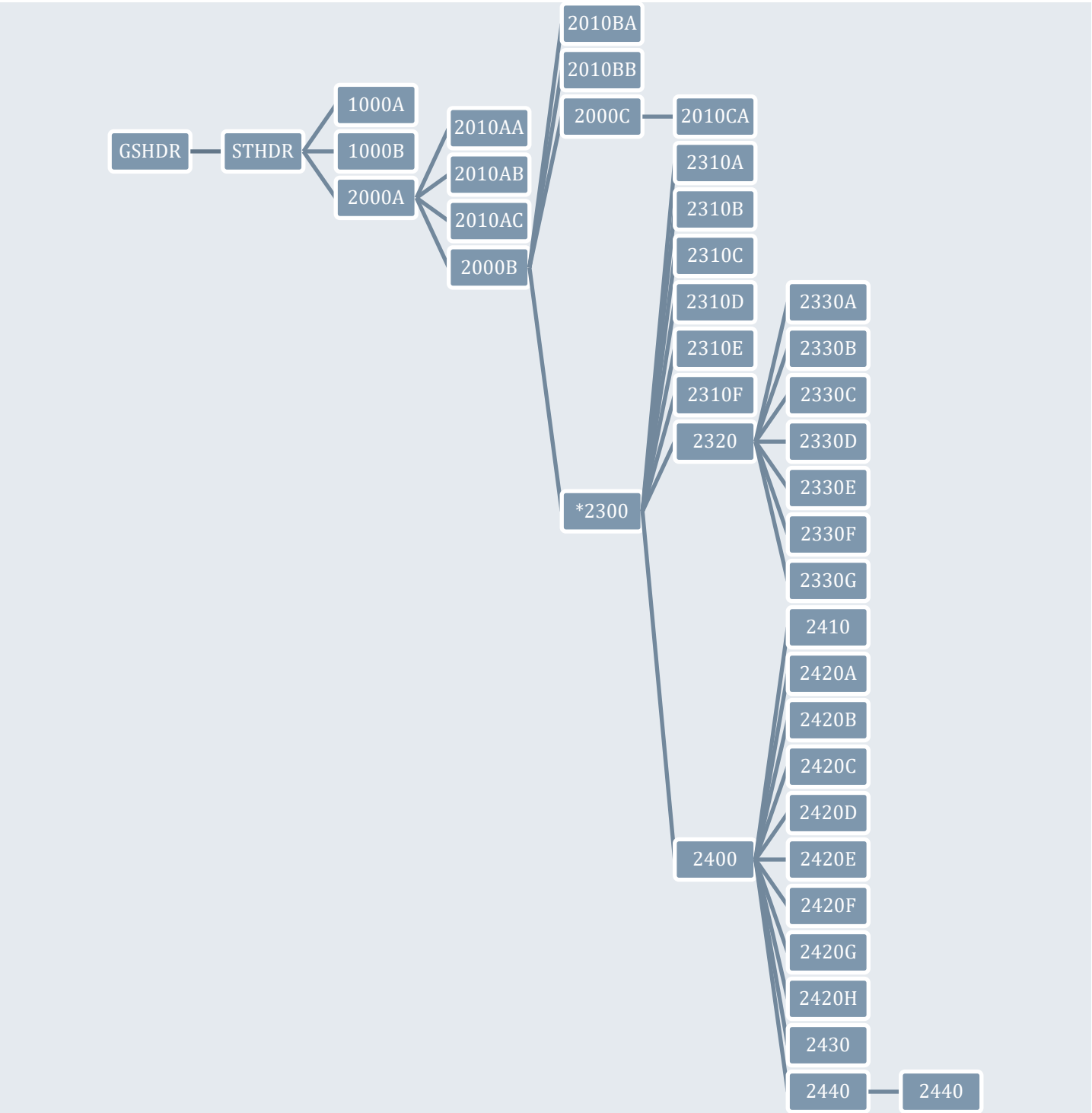
5010_835 / U0 Health Care Claim Payment/Advice



5010_837I / W0 Health Care Claim: Institutional



5010_837P / X0 Health Care Claim: Professional



* The L2300 tables in the 837 I and 837 P specifications have *two* parent indexes – one for 2000B as shown in these diagrams, and an *additional*, *optional* PAR_2000C_IX field that relates the claim to a specific iteration of the patient loop. If there are no L2000C patient loops, then this field can be left null.

Appendix C: Specification Codes

Specification Tag	Transaction Set / Addenda Level
M0	5010_270
M1	5010_270_A1
N0	5010_271
N1	5010_271_A1
O0	5010_276
P0	5010_277
P5	5010_277CA
P7	5010_824
Q0	5010_278_REQ
R0	5010_278_RESP
S0	5010_820
S1	5010_820_A1
S5	5010_820X
T0	5010_834
T1	5010_834_A1
U0	5010_835
U1	5010_835_A1
W0	5010_837I
W1	5010_837I_A1
W2	5010_837I_A2
X0	5010_837P
X1	5010_837P_A1

The table at left shows all of the specification codes possible. The specification tag consists of a letter and a number. Together, they define a specific transaction set and addenda level.

The specification tag is used for CGIF2-formatted Flat, HDB, and XML data projections, and is used to tell the SERENEDI engine what specification these mappings belong to.

For Flat and HDB projections, the specification code is found in the first ISA mapping. For XML, it's attached to the document root node.

In the 5010 Implementation Guides, the Addenda indicate relatively minor updates to segments and requirement rules; the loops are unchanged..

Appendix D: Rules Engine

To understand what the Rules Engine (RE) in SERENEDI is used for, it helps to understand the three classes of integrity checks SERENEDI is capable of running on decoded EDI files:

1. Basic Syntax – This is built into the decoding process and will generate messages based on simple syntax checks within the segments and loops. It will check for the existence of mandatory loops, segments, and elements. It will also check for elements containing invalid data and generate messages for all of these errors.
2. Special Segment Rules – Certain segments possess specific messages depending on the role the segment plays. For example, if an ST segment starts a transaction with a certain Transaction Control Number (TCN) and an SE segment ends the transaction with a different TCN, that is a segment rule violation and is flagged as such. If the file is decoded with the "Code Set Validation" option, then it is at this level these code sets are validated:
    a. Claim Adjustment Reason Codes
    b. Claim Frequency Codes
    c. ICD 9 CM Diagnosis
    d. ICD 9 PCS Procedure Codes
    e. ICD 10 CM Diagnosis
    f. ICD 10 PCS Procedure Codes
    g. National Drug Code
    h. Provider Taxonomy Codes
    i. Remittance Advice Reason Codes
    j. State Abbreviations
    k. 5-Digit ZIP Codes
    l. Country Codes
3. Rules Engine – This contains all the logic for the SNIP Type 3-5 integrity checks. When a file is successfully decoded, the SCORE script command sapi-CheckIntegrity can be used to validate these deeper integrity checks and generate messages when integrity violations are found. Currently, only 834, 835, and 837 I & P are supported for these integrity checks, with over 300 different rules. Each rule represents a specific error condition defined in the HIPAA Implementation Guides and is defined by a small programming language called REP (Rules Engine Programming). Individual Boolean REP scripts are called REP Code.

SERENEDI enables you to add custom REP rules. In this way, you can add custom validations to your business processes. Because REP was designed specifically for this purpose, it contains customized operations that make it far easier to create these rules compared to other ways, such as checking SQL elements. Each REP rule is codified in a single line of code that evaluates to a single Boolean expression – if it is true, then the error is flagged and a message is generated.

REP CODE Overview

REP rules can only be run when both the SegPool register and HKey register are loaded, and the HKey represents the decode of that particular SegPool. When these conditions are true, then the SERENEDI Integrity Engine can be invoked using the sapi-CheckIntegrity command. However, if you wish to add custom validations, you can do so *before* this command is invoked by using the sapi-AddIntegrityRule command.

It takes the following parameters:

Argument	Type	Purpose

SpecCd	String	This represents all the specifications that trigger the rule, without spaces or separator characters. The loop specified in LoopNm must be present in all the specifications listed. Example: W0W1W2 means the rule triggers on all 837 I specifications, including A1 and A2.
LoopNm	String	This is the loop within the specification this rule is bound to. This rule is executed for each instance of this loop within the HKey register. The loop name should be prefixed by L if it is not a part of the envelope loops.
RuleOrder	Integer	This is the numeric order of the execution. For custom rules, this must be 10000 or higher.
RuleCode	String	This is the REP Code of the rule.
Message	String	This is the message added to the message log when the REP is fired. If there were segment mappings within the REP Code that bind to this same loop, then the messages will be tied to that segment. Otherwise, the message will be tied to iterations of this loop that fire the message.

Before diving into actual examples of REP Codes, we need to distinguish the way CGIF2 maps are used within this syntax. In the CGIF2 mapping system, SERENEDI can define elemental maps containing specific loop iterations and specific segment iterations. For example, the 837 I 2320 loop can have 10 loop iterations, each referenced with a different loop iteration prefix. With REP Codes, all numeric loop iterations are ignored, so a mapping that explicitly defines the first iteration of a 2320 loop will also "hit" on the second and third iterations. Furthermore, segment iterations are normally *also* ignored, but this can be overridden with certain commands.

REP Codes are a single Boolean statement using a simple, common syntax. Behind the scenes, this syntax is *tokenized* into nodes that represent literals of different data types and commands. These nodes are an internal format that allows the Boolean expression to be evaluated quickly every time the associated loop for this REP Code occurs in the EDI file. There are three types of nodes: Literal nodes, Map nodes and Operation nodes. Literal nodes can have one of five types – String, Date/Time, Integer, Double, and Boolean. Map nodes are strongly typed references to data elements that can occur in the same loop the rule is bound to, or in a parent or child loop. Operation nodes will resolve to one of these values, depending on the operation.

REP CODE Example

To get an idea of what you can do with REP Codes, here is an actual integrity rule for the 837 P specification:

SPEC: X0X1 **Loop: L2400** **Order: 330** **Message:** AMT Tax should not exceed Line Item Charge Amount

```
L2400_AMT02_SALES_TAX_AMT.IsPresent && (L2400_AMT02_SALES_TAX_AMT >
L2400_SV102_LIN_ITM_CHG_AMT)
```

Because this REP Code is linked to the Service Line loop within the 837 Professional specification, this rule will be evaluated once for every Service Line in a file. The IsPresent token evaluates the linked map and returns a True if the mapping exists for this Service Line. The && is a Boolean "and" operation, followed by an expression that checks whether the Sales Tax Amount segment exceeds the Line Item Charge Amount.

Because the first mapped element in the REP Code belongs to the AMT segment, and this segment lies on the same loop it is bound to (Loop 2400), any messages generated by this integrity check will be bound to that segment. This is a simple example to demonstrate what REP Codes are and how they form the backbone of the SERENEDI Integrity Engine.

Testing new REP CODES

New REP Code rules can only be added *after* a SegPool has been loaded and decoded to an HKey. Therefore, you will need one of the included seed files to "prime" the environment so new rules can be added. It is possible to output an XML format display of what the parsed REP Code looks like to the internal environment; this is proof that it compiled properly.

You can use the SERENEDI REPL environment *or* the SERENEDI Studio RunBox for this walkthrough – there is a slight variation. For the purposes of this walkthrough, I will focus on the REPL environment. First, bring up the REPL command screen, (TODO: how to do this), then:

```
sapi-SegPoolFromFile -Filename 'C:\serenedi\seed\seed_837p.txt'
sapi-SegPoolToHKey
Write-Host (sapi-AddIntegrityRule -SpecCd "X0X1" -LoopNm "L2400" -RuleOrder 10000 -
RuleCode "L2400_AMT02_SALES_TAX_AMT.IsPresent && (L2400_AMT02_SALES_TAX_AMT >
L2400_SV102_LIN_ITM_CHG_AMT)" -RuleMessage "AMT Tax should not exceed Line Item Charge
Amount" -ShowXMLTokens)
```

For SERENEDI Studio users only: replace **Write-Host** with **Write-Output**, and you'll see the XML results displayed in the RunBox window in a compact form. There are various ways to get the XML displayed properly – one method is to install the open-source text editor NotePad++, install the XML tools, paste the XML into a new window, and type Ctrl-Alt-Shift-B.

The output from this command follows:

```
<Storage Data="L2400_10000:AMT Tax should not exceed Line Item Charge Amount">
  <AND ND="L">
    <MapDoesExist ND="L">
      <Map ND="L">L2400_AMT02_SALES_TAX_AMT</Map>
    </MapDoesExist>
    <GT ND="R">
      <Map ND="L">L2400_AMT02_SALES_TAX_AMT</Map>
      <Map ND="R">L2400_SV102_LIN_ITM_CHG_AMT</Map>
    </GT>
  </AND>
</Storage>
```

Now, run the command **sapi-CheckIntegrity**. On the seed 837 P file, this will not fire the trigger because it doesn't have the AMT segment. Go ahead and make two copies of the SEED_837P file:

```
C:\serenedi\pipeline\seed_837p_pos_test.txt
C:\serenedi\pipeline\seed_837p_neg_test.txt
```

Edit the first file and place a new segment, **AMT*T*294~**, between the DTP*472 segment and the LX*2 segment. This is the *positive test*, which replicates the condition being tested and shows that the check functions. On line 818, change the SE*815 segment to SE*816 to reflect the new segment count.

Edit the second file and place a new segment, **AMT*T*1~**, in the same place as before. This is the *negative test*, which proves that the new integrity rule doesn't fire when the criteria do not match. Also, change the segment count as above.

Back in the REPL environment:

```
sapi-Reset
```

```
sapi-SegPoolFromFile -Filename 'C:\serenedi\pipeline\seed_837p_neg_test.txt'
sapi-SegPoolToHKey
sapi-AddIntegrityRule -SpecCd "X0X1" -LoopNm "L2400" -RuleOrder 10000 -RuleCode
"L2400_AMT02_SALES_TAX_AMT.IsPresent && (L2400_AMT02_SALES_TAX_AMT >
L2400_SV102_LIN_ITM_CHG_AMT)" -RuleMessage "AMT Tax should not exceed Line Item Charge
Amount"
sapi-CheckIntegrity
```

An integrity check doesn't result in any messages, which is good because this is the negative test.

Now, change the above script to replace **seed_837p_neg_test** with **seed_837p_pos_test**, and run the whole script again. We get different results now:

```
SerenediREPL:
sapi-CheckIntegrity
IL2400_10000|L2400_10000:AMT Tax should not exceed Line Item Charge Amount||27|0
IL2400_330|L2400_330:AMT Tax should not exceed Line Item Charge Amount||27|0
```

Since this example is basically copying an existing rule, it's no surprise we are seeing the same error twice. This is also a very simple example; the REP Code syntax is capable of far more complex operations than demonstrated here.

REP CODE Token Library

Token(s)	Value	Operation
( )		The parenthesis tokens guide the tokenization of the REP Code expression. Enclose every Boolean expression with its own set of parentheses to ensure the operations occur in the correct order.
&&	Boolean	Logical AND
||	Boolean	Logical OR
!	Boolean	This precedes a tokenized expression and reverses the Boolean result of that expression.
+	Integer String Double	This operator takes on the type of its right node and adds the two expressions together. Integers and Doubles will be added, and Strings will be concatenated.
-	Integer Double DateTime	This operator takes on the type of its right node and subtracts the expressions. Integers and Doubles will be subtracted. For DateTime expressions, the total days difference will be divided by 365.25, converted to an Integer, and returned as an Integer type. **Example:** `((STHDR_BGN03_TS_CRTN_D8 – L2100A_IL_DMG02_MBR_DOB) < 19)` The above 834 REP Code expression will resolve to True if the difference between the transaction set creation date and the member's date of birth – the age of the member at the time the transaction was generated – is less than 19 years.
< <= > >=	Boolean	These operators will do a value comparison between the left and right nodes. They will work on Integer, Double, and DateTime values.
>=	Boolean	When this compares two strings, then the left-hand string is discarded and the right-hand node is evaluated; it returns True if there is a match in the internal text pool generated by the = operator below, and False otherwise.

!=	Boolean	This is a Boolean “Not Equal” operation, and will work on DateTimes, Doubles, Integers, and Strings. Both strings will be trimmed of leading and trailing spaces prior to the comparison.
==	Boolean	This is a Boolean “Equal To” operation, and functions similar to the != operation immediately above.

=		The Variable Assignment operator, Equals, plays two roles. In one role, it passively links a variable defined by a string literal to a series of nodes, which are evaluated by the Fetch token, described later.
		If the right-hand expression evaluates to a string expression, then it stores the results of the expression in an internal storage area.
		
		Example (Passive Link):
		`‘varMedicare’ = ((L2000B_SBR09_CLM_FIL_IND_CD == ‘MA’) || (L2000B_SBR09_CLM_FIL_IND_CD == ‘MB’))`
		
		The above 837 P mapping tied to the L2000B loop stores a Boolean value depending on whether the Claim Filing Indicator is set to Medicare Part A or Part B. This variable can then be referenced by other REP Code rules, such as here:
		
		`L2300_REF_FAC_ID.IsPresent && (!’varMedicare’.Fetch)`
		
		Here, this rule tied to Loop 2300 in the same spec references the Medicare variable and fires an error if the Care Plan Oversight ID is present in non-Medicare claims.
		
		Example (String Accumulator):
		
		<u>834, Loop 2000, Order 160</u>
		`'varSub' = L2000_INS01_MEM_IND + L2000_INS03_MAINT_TYP_CD + L2000_REF_SUB_NR`
		
		<u>834, Loop 2000, Order 170</u>
		`('varSub' >= ('N021' + L2000_REF_SUB_NR)) && (L2000_INS01_MEM_IND == 'Y') && (L2000_INS03_MAINT_TYP_CD == '021')`
		In the first REP Code, the right-hand expression evaluates to a string value, and the ‘varSub’ string literal is ignored – the results are stored in a single internal text pool. This can be reset using the Clear command, but will otherwise accumulate values. Since this expression does not evaluate to a Boolean True, there is no Rule Message associated with this rule – it works in conjunction with another REP Code.
		The second REP Code uses the VarFetch function and the >= token, and returns a Boolean True if the string expression exists within the text pool. In this business context, it means that a dependent with the same subscriber number as the current loop preceded this one, which is a violation of the HIPAA Integrity rules.

Clear		This completely resets all variables.
‘XYZ’	String	String literals are enclosed in single quotes. When passing commands in PowerShell Core, it’s important to use double quotes around all expressions.
123	Integer	Integer literals can be plus or minus, and have no decimal point.
#19000101#	DateTime	DateTime literals are always 8 digits (Year – Month – Day format) surrounded by hash characters.

0.00	Double	Double literals will always have a decimal point.
True	Boolean	Boolean literal
False	Boolean	Boolean literal
Fetch		This is attached to a string literal of a variable name, and retrieves the value stored in that value. This node takes on the type of whatever the variable was assigned to.
EvalExact		This is attached to a mapping with explicit segment repeats. Normally, REP Code mappings match on the first occurrence within a loop without regard to loop numeric iterations or segment iterations. EvalExact allows checking on explicit iterations. **Example:** `L2100_02PER02_CLM_CON_NM.IsPresent && (L2100_02PER02_CLM_CON_NM.EvalExact == L2100_PER02_CLM_CON_NM)` This 835 REP Code example shows how to ensure the second iteration if the name in the 2100 PER segment doesn't match in the first iteration.
Length	Integer	This returns the length of the attached string expression. **Example:** `L2300_CLM01_PT_CTL_NR.Length > 20` This evaluates to True when the length of the Patient Control Number exceeds 20 characters.
%= **%!**	Boolean	These tokens represent the MapIsMatched and MapNotMatched operations. This is a *set* operation in that it takes a single value evaluated in the left-hand node and assesses it against a set of values represented in the right-hand node. The right-hand node is usually a mapping that is higher within the loop hierarchy tree, usually in a loop that repeats. This operation will gather all values that satisfy the right-hand map in relation to the left-hand map; if a single match is found, the MapIsMatched will return a True result, and will otherwise return False. Likewise, MapNotMatched will return True unless a match is found. **Example:** `L2420E_REF0402_PYR_ID.IsPresent && (L2420E_REF0402_PYR_PRI_ID %! L2330B_01_NM109_HCFA_PLAN_ID) && (L2420E_REF0402_PYR_PRI_ID %! L2330B_01_NM109_PAYR_ID)` In the above example from the 837 Professional spec, this rule checks to see if the Loop 2420E Other Payer Primary Identifier matches any of the 2330B Payer IDs or HCFA Plan IDs in any of the iterations of the 2320 loops that lie relative to this 2420E loop within the same claim.
IsPresent **NotPresent** **IsPresentExact** **NotPresentExact**	Boolean	These operations will return True if the attached map exists, and False otherwise. This is also a set-based operation and can operate in different loops then the one the rule is bound to, similar to the example above for the %= token. The operations normally do not check for segment iterations, so for segments that can have multiple repetitions, it will match regardless of the iteration. If the Exact operations are used, then the segment iteration *is* used for the evaluation.
IsPOBox	Boolean	This will evaluate the attached map and return True if it appears to be formatted like a PO Box address.

IsChild **NotChild**	Boolean	This command is used in deeper-level mappings relative to the loop bound to the rule to see if that loop exists as a child to the bound loop. It must be bound to an element that is mandatory for the loop in question. It will return True if that loop is a child loop of the bound loop (existing lower on the hierarchy tree), False otherwise for the IsChild token, and the reverse of that for the NotChild token.
***=**		This token is the Dynamic Assign operator. The string literal to the left of the token is resolved to be the *name* of a new variable, which is assigned a value with the expression to the right of the token. This value can later be retrieved within another REP Code via the DynFetch token.
DynFetch		This pulls values that were assigned via the Dynamic Assignment operator. It's useful when you need to store values across various iterations of loops and make comparisons.

VSum
HSum
	Double	**VSum**, or Vertical Summation, is an operation in which a higher-order loop needs to sum up values lying within standard child loops.
		HSum, or Horizonal Summation, is an operation in which a higher-order loop needs to sum up values lying within loops parented to non-standard iterated loops. This includes loops with a loop count of less than 25, value-defined loops, and anything not defined as a standard loop.
		
		Example:
		`L2430_01_SVD02_SVC_LIN_PD_AMT.IsChild &&`
		`(L2400_SV102_LIN_ITM_CHG_AMT !=`
		`(L2430_01_SVD02_SVC_LIN_PD_AMT.VSum +`
		`L2430_01_01CAS03_ADJ_AMT.VSum +`
		`L2430_01_01CAS06_ADJ_AMT.VSum +`
		`L2430_01_01CAS09_ADJ_AMT.VSum +`
		`L2430_01_01CAS12_ADJ_AMT.VSum))`
		
		In the above REP Code, the VSum command is used to dynamically sum up the Adjustment Amounts throughout the 2430 loop. Since this command is bound to the 837 Professional 2400 Claim Line loop, it sets up a comparison between the Line Item Charge Amount and the 2430 Claim Adjustment segments.

PRSum	Double	This is similar to a VSum operation as above, except it will match *only* in mappings that belong to CAS*PR segments.