

SERENEDI

HEALTHCARE INTEGRATION PLATFORM
SNOWFLAKE EDITION

© 2026 CHIAPAS EDI TECHNOLOGIES, INC. ALL RIGHTS RESERVED.

REVISION 202650407

Contents

Licensing	4
User Manual	5
Introduction	5
Installation	7
Quick-Start Script	7
Step-by-Step Walkthrough	8
System Overview	12
HEARTBEAT Main Stored Procedure	15
Sample Stored Procedures	15
Chiapas Gate Intermediate Format, Version 2	18
Encoding vs. Decoding	25
Defaulted Scaffold Elements	26
Encoding/Decoding	27
Functionality Walkthrough	30
Phase 1: Generate Seed EDI Files (EDI_FROM_BIN)	31
Phase 2: Compliance Checking (INTEGRITY)	32
Phase 3: Decode EDI to CSV (CSV_FROM_ED I)	33
Phase 4: Decode EDI to XML (XML_FROM_ED I)	33
Phase 5: Decode EDI to Flat BIN Tables (EDI_TO_BIN)	33
Phase 6: Decode EDI to Hierarchical Database (EDI_TO_HDB)	34
Phase 7: Round-Trip — CSV to EDI (CSV_TO_ED I)	34
Phase 8: Round-Trip — XML to EDI (XML_TO_ED I)	35
System Architecture	36
Trigger Scan	37
Poll Task	37
Process SYS_MSQ	37
Workflows	38
TECHNICAL REFERENCE	44
Supported Transaction Set Identifiers	44
SCHEMA REFERENCE	44
Sample Data Tables	55
Creating Outbound Transactions	58

SERENEDI Architecture	91
Data Ingestion	92
System Configuration	94
Integrity Validations	94
SERENEDI Core Schema	95
SERENEDI Incident Response Plan	100

Licensing

SERENEDI is Copyright © 2026 Chiapas EDI Technologies, Inc. All Rights Reserved.

For consumer's obligations to Chiapas EDI Technologies, Inc., see Snowflake's Standard Agreement for Marketplace Products at:

<https://www.snowflake.com/en/legal/optional-offerings/offering-specific-terms/snowflake-marketplace/standard-agreement/>

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

Chiapas EDI Technologies, Inc. would like to express appreciation to the authors and publishers of the following libraries:

.NET 5 and .NET 8, © Microsoft 2021, under the MIT License

User Manual

Introduction

The Challenge of Healthcare Integration

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 is eligibility. If a provider group has a three-month-old snapshot of eligibility from the payer, it could be sending claims for a patient who was disenrolled two months ago. These claims would be denied, and the provider would be left 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 early days of healthcare IT, this integration was handled with flat files or CSV files 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 attempts to standardize file formats, little to no legislation supported them and therefore they lacked incentive. By the late 1990s, the healthcare industry was drowning in millions of proprietary data formats, and even small changes in business requirements demanded heavy investment in development time.

Enter the Health Insurance Portability and Accountability 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 fixed file formats for specific business-to-business transactions and authored HIPAA Implementation Guides. These guides became the governing standard for what each file would look like and what data it could contain.

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

These hierarchical data formats differ vastly from the CSV or flat files that came before. Furthermore, the tools and technologies available to work with these complex, hierarchical formats have a steep learning curve and demand significant development time. This is the problem SERENEDI was built to solve.

SERENEDI for Snowflake

SERENEDI is a third-generation healthcare integration platform—building on the 2003 product Chiapas Version 1, which became the 2012 product Chiapas EDI Enterprise, which in turn evolved into the 2020 release of SERENEDI on Windows. It is designed specifically to address complexity of HIPAA X12 EDI, and has been reengineered to focus on the requirements of the Snowflake platform. The translation engine runs in Snowpark Container Services (SPCS), Snowflake's managed container infrastructure, and all data flows through Snowflake stages and tables.

Security Architecture

SERENEDI runs as a combination of stored procedures and a binary SPCS Container image. No consumer data leaves the Application, with all errors are logged directly to consumer stages. SERENEDI does not utilize any sort of telemetry. Internally, the SERENEDI Container image contains binary representations of the X12 transaction sets (used under license

– see our logo at <https://x12.org/products/licensing-program/partners>) as well as the codesets used for EDI integrity validations. These codesets are updated directly from their various sources twice a year, with the SERENEDI application updated on or around the first of February and August each year. SERENEDI operates with the minimal permissions needed to function.

Mapping Technology

SERENEDI’s proprietary technology *projects* EDI transactions directly into four formats: XML, CSV files, database flat tables, or database hierarchical tables. This projection is bidirectional, and as long as SERENEDI’s mapping rules are maintained and the file is completely HIPAA compliant, it can generally reproduce the original EDI transaction character for character. These projections are completely automated, “black box” operations that are built into SERENEDI.

The core mapping engine and its unique schema is a balanced middle ground between two competing requirements: maintain the ability to encode every valid EDI element and scenario, while presenting the data in human readable form through projecting the underlying healthcare information to elements closely named to the data it represents, such as: L2010BA_NM109_MEM_ID_NR (Subscriber Member ID) or L2300_REF_PRIOR_AUTH_NR (837 Claim Prior Authorization Number). SERENEDI’s technology manages the metadata to the maximum possible extent, making the actual process of developing and maintaining the trading partner data flows much easier. This technology has been implemented for 18 Healthcare related transactions: 5010 270, 271, 276, 277, 277 CA, 278 NOT, 278 ACK, 278 REQ, 278 RESP, 820, 820X, 824, 834, 835, 837 D, 837 I, 837 P, and 999.

SERENEDI is event-driven. You stage EDI files to a Snowflake stage, and SERENEDI detects and processes them automatically through its BIZ_EVENT orchestration system. Eight event types drive the platform’s workflows:

CSV_FROM_EDI and CSV_TO_EDI – Transform EDI transactions to and from CSV format.

XML_FROM_EDI and XML_TO_EDI – Transform EDI transactions to and from XML format.

EDI_TO_BIN and EDI_FROM_BIN – Load EDI data into denormalized Flat BIN tables, or generate EDI from them.

EDI_TO_HDB – Load EDI data into the normalized Hierarchical Database for efficient storage and complex querying.

INTEGRITY – Run the compliance rules engine against EDI files to produce HTML integrity reports measuring conformance with HIPAA Implementation Guide rules.

The Flat BIN system offers highly accessible, denormalized data that is easy to query with standard SQL. The Hierarchical Database trades some of that accessibility for storage efficiency and structural fidelity, storing each EDI loop in its own table with parent-child relationships preserved. Both systems output to standard Snowflake tables that can be joined, filtered, and analyzed with the full power of Snowflake SQL. The maps are defined both in supplied HTML files as well as a supplied CSV file providing over 13,000 maps.

The compliance rules engine measures EDI files against the rules defined in the HIPAA Implementation Guides for a subset of the supported transactions: 270/271, 834, 835, 837P, and 837I, and covers SNIP 1-5 integrity checks for these transactions.

To help offset the learning curve for creating outbound EDI transactions, the SERENEDI distribution ships with sample data and reference procedures for generating seed files for each of the 14 supported 5010 transaction sets. For inbound EDI, the default pipeline transforms files staged to your Snowflake stage into queryable Snowflake tables—and as new fields are encountered in incoming data, the BIN tables are automatically extended with new columns to hold them.

We hope this user manual clearly presents the capabilities of the SERENEDI engine in its Snowflake incarnation, 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.

Installation

SERENEDI is a Snowflake Native App. Installation creates the application within your Snowflake account, provisions the resources it needs to run, and configures a consumer-owned database where your EDI files and processed output will reside. No software is installed outside of Snowflake.

Prerequisites

Before you begin, confirm that you have the following:

Requirement	Details
Snowflake Account	An active Snowflake account with access to the Snowflake Marketplace.
ACCOUNTADMIN Role	The initial setup steps require ACCOUNTADMIN privileges to install the application and grant the necessary account-level permissions.
Compute Pool Privileges	The account must allow creation of compute pools for Snowpark Container Services (SPCS). SERENEDI's containers run within these pools.
Warehouse	A Snowflake virtual warehouse is required for SQL operations. SERENEDI will create or reference one during setup.

Quick-Start Script

The following script performs a complete SERENEDI installation. This script is also found in the README. Expert users can run this as a single block from a Snowsight SQL worksheet. Each step is explained in detail in the sections that follow.

Note on roles: This script creates a dedicated SERENEDI_ROLE for day-to-day SERENEDI operations. If your organization prefers to use an existing role, replace every occurrence of SERENEDI_ROLE in this script with your role name and remove the CREATE ROLE and GRANT ROLE statements.

```
-- =====
-- SERENEDI for Snowflake: Installation Script
-- Replace SERENEDI_ROLE with your own role if preferred.
-- =====

-- Step 1: Install the application (skip if installed via Marketplace)
-- USE ROLE ACCOUNTADMIN;
-- CREATE APPLICATION SERENEDI FROM ...;

-- Step 2: Grant account-level privileges
USE ROLE ACCOUNTADMIN;
CALL SERENEDI.MAIN.GRANT_CALLBACK(
  ARRAY_CONSTRUCT('CREATE COMPUTE POOL', 'CREATE WAREHOUSE')
```

```

);

-- Step 3: Create the operating role (optional – use your own role instead)
CREATE ROLE IF NOT EXISTS SERENEDI_ROLE;
GRANT ROLE SERENEDI_ROLE TO ROLE SYSADMIN;

-- Step 4: Create the consumer database and transfer ownership
CREATE DATABASE IF NOT EXISTS SERENEDI_DATA;
GRANT OWNERSHIP ON DATABASE SERENEDI_DATA TO ROLE SERENEDI_ROLE;

-- Step 5: Switch to the operating role (use the role defined during installation)
USE ROLE SERENEDI_ROLE;

-- Step 6: Create schemas and stages
CREATE SCHEMA IF NOT EXISTS SERENEDI_DATA.STAGES;
CREATE SCHEMA IF NOT EXISTS SERENEDI_DATA.OUTPUT;

CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.EDI_IN DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.EDI_OUT DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.CSV_IN DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.CSV_OUT DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.XML_IN DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.XML_OUT DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.ERR DIRECTORY = (ENABLE = TRUE);

-- Step 7: Grant the application access to consumer resources
GRANT USAGE ON DATABASE SERENEDI_DATA TO APPLICATION SERENEDI;
GRANT USAGE ON SCHEMA SERENEDI_DATA.STAGES TO APPLICATION SERENEDI;
GRANT USAGE ON SCHEMA SERENEDI_DATA.OUTPUT TO APPLICATION SERENEDI;
GRANT READ, WRITE ON ALL STAGES IN SCHEMA SERENEDI_DATA.STAGES TO APPLICATION SERENEDI;
GRANT CREATE TABLE ON SCHEMA SERENEDI_DATA.OUTPUT TO APPLICATION SERENEDI;

-- Step 8: Grant the application role to the operating role
GRANT APPLICATION ROLE SERENEDI.APP_PUBLIC TO ROLE SERENEDI_ROLE;

-- Step 9: Configure the application
DELETE FROM SERENEDI.MAIN.APP_CONFIG
WHERE CONFIG_KEY IN (
  'CONSUMER_STAGES_PREFIX',
  'CONSUMER_OUTPUT_PREFIX',
  'CONSUMER_ROLE'
);

INSERT INTO SERENEDI.MAIN.APP_CONFIG (CONFIG_KEY, CONFIG_VAL) VALUES
('CONSUMER_STAGES_PREFIX', 'SERENEDI_DATA.STAGES'),
('CONSUMER_OUTPUT_PREFIX', 'SERENEDI_DATA.OUTPUT'),
('CONSUMER_ROLE', CURRENT_ROLE());

```

Step-by-Step Walkthrough

The previous instructions are a quick way to finalize the installation of SERENEDI. Use the following instructions if you need a more detailed explanation of each step.

Step 1: Install the Application

Locate SERENEDI in the Snowflake Marketplace and click Get to install it. Snowflake will create the SERENEDI application in your account. Once the installation is complete, you will have a new application object named SERENEDI visible under Snowsight → Data Products → Apps.

Marketplace Flow — Subject to Change

The exact Marketplace installation experience may vary depending on Snowflake's current UI and any consent screens presented during installation. If Snowflake prompts you to approve account-level privileges (such as creating compute pools or warehouses) during the Marketplace install, approve them and skip the GRANT_CALLBACK step below.

Step 2: Grant Account Privileges

SERENEDI needs permission to create a compute pool (for its SPCS containers) and a warehouse (for SQL operations). Run the following as ACCOUNTADMIN:

```
USE ROLE ACCOUNTADMIN;  
CALL SERENEDI.MAIN.GRANT_CALLBACK(  
  ARRAY_CONSTRUCT('CREATE COMPUTE POOL', 'CREATE WAREHOUSE')  
);
```

This is the only step that requires ACCOUNTADMIN. All subsequent steps can be performed with a less-privileged role.

Note

If the Marketplace installation already prompted you to grant these privileges through a consent dialog, this step is unnecessary. You can verify by checking whether the application can already create compute pools and warehouses.

Step 3: Create the Consumer Database

SERENEDI follows a consumer-owned data pattern: your EDI files, processed output, and BIN tables all reside in a database that you own and control, separate from the application itself. Create this database and its two required schemas:

```
CREATE DATABASE IF NOT EXISTS SERENEDI_DATA;  
  
CREATE SCHEMA IF NOT EXISTS SERENEDI_DATA.STAGES;  
CREATE SCHEMA IF NOT EXISTS SERENEDI_DATA.OUTPUT;
```

SERENEDI_DATA.STAGES holds the file stages where you will place inbound files and retrieve outbound results.

SERENEDI_DATA.OUTPUT holds the Snowflake tables that SERENEDI creates when processing EDI into BIN or HDB format.

Why a Separate Database?

Snowflake Native Apps cannot create objects outside their own application database without explicit grants. The consumer-owned database pattern keeps your data under your control, ensures the application can be cleanly uninstalled without orphaned objects, and lets you apply your own access policies and role hierarchy to the processed data.

Step 4: Create Stages

Stages are the file-system interface between your environment and SERENEDI. Create the following stages within the STAGES schema:

```
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.EDI_IN
  DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.EDI_OUT
  DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.CSV_IN
  DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.CSV_OUT
  DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.XML_IN
  DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.XML_OUT
  DIRECTORY = (ENABLE = TRUE);
CREATE STAGE IF NOT EXISTS SERENEDI_DATA.STAGES.ERR
  DIRECTORY = (ENABLE = TRUE);
```

Stage	Purpose
EDI_IN	Place inbound EDI files here for decoding (EDI → CSV, XML, BIN, HDB, or integrity checking).
EDI_OUT	SERENEDI writes encoded EDI files here (CSV → EDI, XML → EDI, BIN → EDI).
CSV_IN	Place inbound CSV files here for encoding to EDI.
CSV_OUT	SERENEDI writes decoded CSV output here.
XML_IN	Place inbound XML files here for encoding to EDI.
XML_OUT	SERENEDI writes decoded XML output here.
ERR	SERENEDI writes error output and diagnostics here.

Step 5: Grant the Application Access

The SERENEDI application needs permission to read and write to the stages you created, and to create tables in the OUTPUT schema for BIN and HDB results:

```
GRANT USAGE ON DATABASE SERENEDI_DATA TO APPLICATION SERENEDI;
GRANT USAGE ON SCHEMA SERENEDI_DATA.STAGES TO APPLICATION SERENEDI;
GRANT USAGE ON SCHEMA SERENEDI_DATA.OUTPUT TO APPLICATION SERENEDI;

GRANT READ, WRITE ON ALL STAGES IN SCHEMA
  SERENEDI_DATA.STAGES TO APPLICATION SERENEDI;

GRANT CREATE TABLE ON SCHEMA
```

```
SERENEDI_DATA.OUTPUT TO APPLICATION SERENEDI;
```

You should also grant the application's public role to the role you will use for day-to-day SERENEDI operations. This allows that role to call SERENEDI's stored procedures and view its status tables:

```
GRANT APPLICATION ROLE SERENEDI.APP_PUBLIC TO ROLE <your_role>;
```

Step 6: Configure the Application

Finally, tell SERENEDI where to find your consumer database. These configuration values are stored in the application's APP_CONFIG table:

```
DELETE FROM SERENEDI.MAIN.APP_CONFIG
WHERE CONFIG_KEY IN (
  'CONSUMER_STAGES_PREFIX',
  'CONSUMER_OUTPUT_PREFIX',
  'CONSUMER_ROLE'
);

INSERT INTO SERENEDI.MAIN.APP_CONFIG
(CONFIG_KEY, CONFIG_VAL) VALUES
('CONSUMER_STAGES_PREFIX', 'SERENEDI_DATA.STAGES'),
('CONSUMER_OUTPUT_PREFIX', 'SERENEDI_DATA.OUTPUT'),
('CONSUMER_ROLE', CURRENT_ROLE());
```

CONFIG_KEY	Purpose
CONSUMER_STAGES_PREFIX	The fully qualified schema path where SERENEDI will look for your file stages. Must match where you created them in Step 4.
CONSUMER_OUTPUT_PREFIX	The fully qualified schema path where SERENEDI will create BIN and HDB tables.
CONSUMER_ROLE	The Snowflake role SERENEDI should use when interacting with your consumer database. Set to CURRENT_ROLE() to use whatever role you are running the install with.

Verifying the Installation

After completing the steps above, you can verify that SERENEDI is correctly installed by confirming the following:

1. The SERENEDI application is visible in Snowsight under Data Products → Apps.
2. The SERENEDI_DATA database exists with STAGES and OUTPUT schemas.
3. All seven stages (EDI_IN, EDI_OUT, CSV_IN, CSV_OUT, XML_IN, XML_OUT, ERR) are present in SERENEDI_DATA.STAGES.

4. The APP_CONFIG table contains the three configuration keys shown above:

```
SELECT * FROM SERENEDI.MAIN.APP_CONFIG;
```

Once verified, SERENEDI is ready to process EDI. The next section covers essential topics before proceeding to a full walkthrough of the SERENEDI functionality.

System Overview

After installation, your Snowflake account contains two databases that make up the SERENEDI environment: the SERENEDI application database, which contains the engine, its configuration, and sample data; and the SERENEDI_DATA consumer database, which contains your stages and processed output. This section introduces the key objects in each database so you know where to look when working with SERENEDI. See **SERENEDI Architecture** to see the schema and relations between the core SERENEDI tables.

The SERENEDI Application Database

The SERENEDI application database is owned and managed by the application itself. You interact with it by calling its stored procedures and querying its tables. With the exception of APP_CONFIG (which holds your configuration) and BIZ_EVENT (where you submit work), most objects in this database are managed by the engine on your behalf. Note there are a number of fields not used in the Snowflake version of SERENEDI; these will be documented later in the technical reference.

Procedure	Transaction
USP_270_EXTRACT	270 — Eligibility Inquiry
USP_271_EXTRACT	271 — Eligibility Response
USP_276_EXTRACT	276 — Claim Status Request
USP_277_EXTRACT	277 — Claim Status Response
USP_277CA_EXTRACT	277CA — Claim Acknowledgment
USP_278_REQ_EXTRACT	278 Request — Prior Authorization Request
USP_278_RESP_EXTRACT	278 Response — Prior Authorization Response
USP_820_EXTRACT	820 — Payment Order/Remittance Advice

managed by the application itself. You interact with it by calling its stored procedures and querying its tables. With the exception of APP_CONFIG (which holds your configuration) and BIZ_EVENT (where you submit work), most objects in this database are managed by the engine on your behalf. Note there are a number of fields not used in the Snowflake version of SERENEDI; these will be documented later in the technical reference.

Tables

Table	Purpose
BIZ_EVENT	The work queue. Every unit of work SERENEDI performs is represented as a row in this table. Events are created in two ways: automatically by TRIGGER_SCAN when new files appear in a watched stage subfolder, or manually when you INSERT a row directly. You query this table to check processing status.
BIZ_MSG	Messages generated during event processing, including errors and diagnostics. Each message is linked to its parent BIZ_EVENT row.
BIZ_TRIGGER	Defines the triggers that SERENEDI scans for automatically. Each trigger specifies a stage subfolder to watch, a glob pattern for file matching, and the type of event to generate when new files are detected.
BIN_LOG	Tracks every BIN and HDB item that SERENEDI has processed. Each row records the BIN ID, source filename, BIN type (Flat BIN or HDB), and processing timestamp. Used when encoding EDI back from BIN or HDB storage.
SYS_MSQ	The system message queue. Used internally by the engine for inter-process coordination.

Two Ways to Submit Work

SERENEDI supports two patterns for submitting work, and you will see both throughout this manual.

Automatic: Place files in a watched stage subfolder (such as @SERENEDI_DATA.STAGES.EDI_IN/CSV_FROM_EDI/) and call HEARTBEAT. The HEARTBEAT detects the new files and generates BIZ_EVENT rows automatically.

Manual: INSERT a row directly into BIZ_EVENT with the appropriate parameters. This is used for workflows like EDI_FROM_BIN, where the work is driven by a stored procedure call rather than a staged file.

SERENEDI ships with sample data tables containing fictional (non-PHI) healthcare data. These tables are the data source behind the USP_*_EXTRACT procedures that generate seed EDI files. They serve as both a starting point for testing and a reference for how to structure your own data extracts.

Table	Contents
-------	----------

SAMPL_HEADER	Interchange and functional group header data (sender/receiver identifiers, dates, control numbers).
SAMPL_MEMBER	Member/subscriber demographic information.
SAMPL_PROVIDER	Provider identifiers, taxonomy codes, and addresses.
SAMPL_PROFESSIONAL	Professional claim service line details.
SAMPL_CLAIM	Claim-level data (diagnosis codes, dates of service, billing amounts).
SAMPL_CLAIM_DTL	Claim detail/line item data.

HEARTBEAT Main Stored Procedure

HEARTBEAT is the primary entry point for all SERENEDI processing. When called, it runs one complete orchestration cycle: first, it calls TRIGGER_SCAN to check all watched stage subfolders for new files and generate BIZ_EVENT rows for anything it finds. Then, it processes all pending events in the BIZ_EVENT table by launching SPCS container jobs to execute each transformation.

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

For production deployments where SERENEDI needs to poll the incoming stages unattended, HEARTBEAT accepts an optional numeric parameter that will put it into *persistent* polling mode. Calling HEARTBEAT with a 15 will cause HEARTBEAT to run automatically every fifteen minute without user intervention:

```
CALL SERENEDI.MAIN.HEARTBEAT(15);
```

Be aware that this will cause **continuous utilization of Snowflake resources** unless turned off by calling it with a zero:

```
CALL SERENEDI.MAIN.HEARTBEAT(0);
```

When called without a parameter, it runs a single cycle and returns. This single-cycle mode is what you will use throughout the Functionality Walkthrough.

Sample Stored Procedures

These procedures query the SAMPL_* tables and produce the data extracts that SERENEDI encodes into EDI files. Each one targets a specific 5010 transaction set. You do not call these procedures directly — instead, you reference them in a BIZ_EVENT row, and the engine calls them during processing.

Procedure	Transaction
USP_270_EXTRACT	270 — Eligibility Inquiry
USP_271_EXTRACT	271 — Eligibility Response
USP_276_EXTRACT	276 — Claim Status Request
USP_277_EXTRACT	277 — Claim Status Response
USP_277CA_EXTRACT	277CA — Claim Acknowledgment
USP_278_REQ_EXTRACT	278 Request — Prior Authorization Request
USP_278_RESP_EXTRACT	278 Response — Prior Authorization Response
USP_820_EXTRACT	820 — Payment Order/Remittance Advice

USP_824_EXTRACT	824 — Application Advice
USP_834_EXTRACT	834 — Benefit Enrollment
USP_835_EXTRACT	835 — Claim Payment/Remittance
USP_837I_EXTRACT	837I — Institutional Claim
USP_837P_EXTRACT	837P — Professional Claim

The Consumer Database (SERENEDI_DATA)

The SERENEDI_DATA consumer database is the database you created during installation. SERENEDI reads from and writes to it, but ownership stays with your role. Even if SERENEDI is completely uninstalled from your account, all of your decoded transaction data stays with you and is not deleted. The *stages* are where SERENEDI’s workflows process files. The *output* schema is where SERENEDI stores data for the BIN system, the SQL queryable projection of PHI data.

SERENEDI_DATA.STAGES

This schema holds seven internal stages that serve as the file-system interface between your environment and the SERENEDI engine.

Stage	What to Put Here
EDI_IN	EDI files for decoding or integrity checking.
CSV_IN	CSV files for encoding to EDI.
XML_IN	XML files for encoding to EDI.

Stage	What Appears Here
EDI_OUT	EDI files generated by encoding workflows (CSV→EDI, XML→EDI, BIN→EDI).
CSV_OUT	CSV files generated by decoding workflows (EDI→CSV).
XML_OUT	XML files generated by decoding workflows (EDI→XML).
ERR	Error output and integrity workflow output.

The Subfolder Convention

When you place files into an inbound stage, the subfolder name determines which workflow processes them. SERENEDI's triggers (defined in BIZ_TRIGGER) watch for files in the "IN" folders and compares it files inside the corresponding "OUT" folder. If the "IN" folder has a new file *not* present in the "OUT" folder, SERENEDI pulls it over and generates a BIZ_EVENT for each new file detected. This is the simple way SERENEDI ensures files are not processed twice – if the same filename already exists in the "OUT" folder, the file will simply lay in the "IN" folder unprocessed.

The default triggers expect files in these subfolder paths:

Subfolder Path	Workflow
EDI_IN/CSV_FROM_EDI/	Decode EDI → CSV
CSV_IN/CSV_TO_EDI/	Encode CSV → EDI
EDI_IN/XML_FROM_EDI/	Decode EDI → XML
XML_IN/XML_TO_EDI/	Encode XML → EDI
EDI_IN/EDI_TO_BIN/	Decode EDI → Flat BIN tables
EDI_IN/EDI_TO_HDB/	Decode EDI → Hierarchical Database tables
EDI_IN/INTEGRITY/	Run HIPAA compliance/integrity checking

SERENEDI_DATA.OUTPUT

This schema is empty after installation. As SERENEDI processes EDI files into BIN or HDB format, it creates tables here automatically. For Flat BIN, you will see tables named by transaction set, such as BIN_5010_837P or BIN_5010_835. For HDB, you will see a set of loop-level tables for each transaction set, with a common three letter prefix for each specification. These are standard Snowflake tables that you can query, join, and analyze with any SQL tool.

As the engine encounters new fields in incoming EDI data, BIN tables are automatically extended with new columns to hold them. You do not need to define or maintain these table schemas — SERENEDI manages them for you.

What's Next

With the architecture in view, the next section — Chiapas Gate Intermediate Format, Version 2 – covers SERENEDI's schema system which is the core technology used for projecting EDI files to and from a more human readable form.

Chiapas Gate Intermediate Format, Version 2

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.

Want a 37-minute training course on mapping with SERENEDI?

TODO: New Home for Video

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 well-defined scaffolding segments 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 solely programmed 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 this mapping system is tightly integrated to the specifications, it does not allow for "custom" segments or departures from the HIPAA implementation guides.

In order to better describe CGIF2 and how it works, let's briefly examine how it functions to *project* a hierarchical EDI file into a two dimensional "flat" CSV. In an 837 Professional Claim file, the deepest part of the hierarchy is generally the 2400 Service Line loop – so when we decode this file to a "Flat" set of CGIF2 maps, the last few columns will be service line related maps like L2400_LX01_ASSGD_NR (Assigned Number), L2400_SV10102_HCPCS_CD (Procedure Code), and so on. Each row in this CSV represents a new service line. Then, preceding these columns are the *claim-specific* maps like L2300_CLM01_PT_CTL_NR (Patient Control Number) and so on. These claim-level maps belong to the parent 2300 loop. Their values repeat unchanged across rows until the service lines belong to a different claim.

As the maps describe higher-order levels of hierarchy, they become more constant. The first map in the "Flat" projection scheme not only describes the first two elements of the outer envelope ISA segment, but also ties the maps to a specific transaction set.

Because the HIPAA transaction sets have a lot of flexibility which SERENEDI needs to encapsulate into a determinative schema, SERENEDI categorizes loops into a number of specific categories. As defined within the HIGs, loops often have specific relationships with other loops, and these relationships play a role in the mapping system. Here's an example of two specific loops in the 837 Institutional schema - 5010_837I.html file

The following sections explain how SERENEDI maps elements and binds them to the hierarchy. Here are two loops from the 5010_837I.html file (from the SERENEDI.REFERENCE.DOCS/specs/5010_837I.html):

L2310F - REFERRING PROVIDER NAME

L2310F	NM1	Referring Provider Name		
03		L2310F_NM103_REF_FVR_LNM	String	Referring Provider Last Name
04		L2310F_NM104_REF_FVR_FNM	String	Referring Provider First Name
05		L2310F_NM105_REF_FVR_MNM	String	Referring Provider Middle Name or Initial
07		L2310F_NM107_REF_FVR_SFX	String	Referring Provider Name Suffix
09		L2310F_NM109_REF_FVR_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_FVR_COMM_NR	String	Provider Commercial Number

L2320 - OTHER SUBSCRIBER INFORMATION (Single Iteration)

L2320	SBR	Other Subscriber Information		
01		L2320_xx_SBR01_FVR_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.

CGIF2 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. These loops 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.

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.

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.

ATTCHMNT – This is a special case data type for storing binary data within a 275 attachment, and only occurs there

Examples

A CGIF2 map is subdivided into three sections. The total amount of characters for each map will never exceed 30 characters. 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:

SERENEDI.REFERENCE.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.

SERENEDI.REFERENCE.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_02CAS03_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:

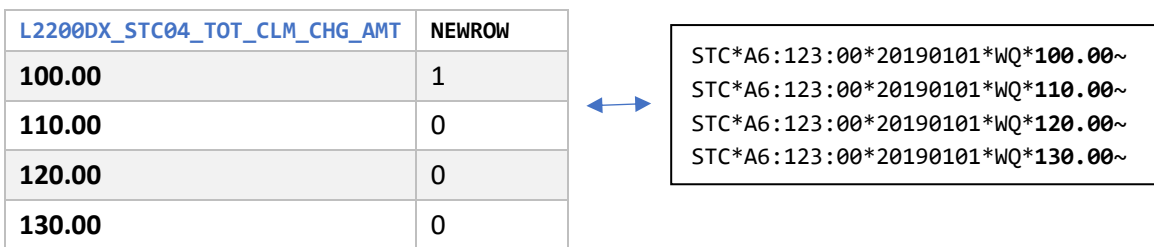
L2200DX - CLAIM STATUS TRACKING NUMBER - STC CUTOFF

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_NFO_EFF_DT	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

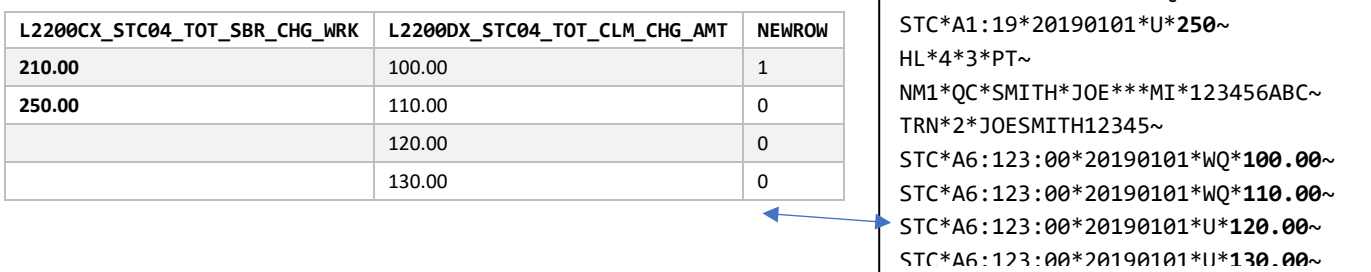
This is an example of a *cutout*. The 2200DX loop does not appear in the HIPAA implementation guides; instead, it's a convention of *cutting out* segments that have infinite repetitions so they can be mapped in a different way. Cutout maps are mapped *vertically* across multiple database rows. If a cutout 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:



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.



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:

DMG05 C056 Comp Race or Ethn Inf X 10
--

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:



Example 7

L2330C_02P3_REF_PVR_COMM_NR

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

SERENEDI.REFERENCE.DOCS/specs/5010_837P_A1.html / Loop 2330C:

L2330C - OTHER PAYER REFERRING PROVIDER (Inherited Loop Iteration & Value Qualified)

Mapping Prefix: L2330C_xxDN - Referring Provider

Mapping Prefix: L2330C_xxP3 - Primary Care Provider

L2330C	NMI	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.

Encoding/Decoding

Encoding CGIF2 Flat to HKey

Within the SERENEDI runtime engine, SERENEDI maintains several registers that it uses to project data from one form to another. The “Flat” register is what is loaded from a Flat database or CSV source; the HKey register is an internal representation of the hierarchical data within a transaction. 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, stored procedure or CSV file, 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 within a single column associated with that loop, 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 processing will continue normally and the deepest standard data-carrying loop will always be considered as “fresh” data for encoding, without regard to the previous row. If the NEWROW 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
O	O	O	O	O	X	1
O	O	O	O	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 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 is dependent on the *order* of the duplicate claims. If they occur in sequential order, then as SERENEDI scans for row differences, it will see *one* claim with *two* sets of service lines.

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. In contrast, if the duplicate claims do not occur sequentially, then it will encode them both as separate claims.

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

Decoding HKey to CGIF2 Flat

The mirror counterpart of encoding an HKey (and by implication, an EDI transaction) is *decoding* an HKey to a Flat. 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." Note that in Snowflake, an unenforced foreign key constraint is automatically generated on each child table to the parent table in order to convey this parent-child relationship to various database tools. It is linked to the (BIN_ID, BIN_IX) fields of the parent table.

The identifiers SERENEDI uses to identify different transaction sets are listed at **Supported Transaction Set Identifiers** in the **TECHNICAL REFERENCE**.

By default, the names of the HDB tables begin with the three digit table name suffix from the previous table, an underscore, and the loop short name. A collection of HDB tables for a specific transaction is called an *HDB tableset*. Every HDB tableset will always have at least an ISA loop.

For example: **CMP_L2320**

The layout of every HDB table begins with four columns:

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
PAR_2000C_IX (int)	Optional field occurring only in 837 D / I / P L2300 Claim loops that links the claim to patient loops

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

L2320_01_SBR01_PYR_RESP_SEQ_NR

For HDB tables, CGIF2 mappings are *very* similar to the Flat interface implementation, with one key difference: all loop iterations belonging to Single Iteration and Value Iteration 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.

WARNING: In the database representation, child loops need to occur in an order that is synchronized with parent loops; in other words, SERENEDI is expecting both the BIN_IX and the PAR_BIN_IX key columns to occur in an incrementing and ascending to be able to register the database rows as valid child loops. This means that you cannot arbitrarily “insert” new children loops by adding new rows to the bottom of the data associated with that BIN_ID; all child loops pertaining to the same PAR_BIN_IX parent loop reference *must* be grouped together in ascending order in relation to the BIN_IX key.

XML Interface

The XML interface projects EDI data to and from XML files. 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. 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>
```

What's Next

Now that we've been exposed to the mapping system and fundamental architecture of SERENEDI, the next section is the hands-on Functionality Walkthrough that demonstrates these different workflows in action.

Functionality Walkthrough

This walkthrough takes you through each of SERENEDI's core workflows using sample data. By the end, you will have generated seed EDI files, decoded them into every supported output format, run compliance checking, and round-tripped them back to EDI. This serves as both a verification of your installation and a hands-on introduction to how SERENEDI operates.

The walkthrough uses three representative transaction sets — 837P (Professional Claim), 835 (Claim Payment/Remittance), and 270 (Eligibility Inquiry) — to demonstrate each workflow. The same patterns apply to all 13 supported transaction sets.

Checking Processing Status

Before diving in, it's important to understand how to tell when SERENEDI has finished processing your work. There are two patterns, depending on the workflow.

For file-based workflows (CSV_FROM_EDl, XML_FROM_EDl, CSV_TO_EDl, XML_TO_EDl, INTEGRITY, EDl_FROM_BIN), processing is complete when the BIZ_EVENT row shows a SUMMARY value of SUCCESS:

```
SELECT BIZ_EVENT_ID, BIZ_TRIGGER_ID, EVENT_CRIT, SUMMARY
FROM SERENEDI.MAIN.BIZ_EVENT
ORDER BY EVENT_DATE DESC;
```

A SUMMARY of SUCCESS means the event has fully completed. If something went wrong, the SUMMARY will reflect the error, and you can check BIZ_MSG for details:

```
SELECT * FROM SERENEDI.MAIN.BIZ_MSG
WHERE BIZ_EVENT_ID = <event_id>
ORDER BY BIZ_MSG_ID;
```

For BIN and HDB ingestion workflows (EDl_TO_BIN, EDl_TO_HDB), BIZ_EVENT reaching SUCCESS means the engine has accepted the file and begun processing, but the data may still be loading into the output tables. To confirm that ingestion is fully complete, check the BIN_LOG:

```
SELECT BIN_ID, BIN_FILENAME, BIN_TYPE, BIN_STATUS
FROM SERENEDI.MAIN.BIN_LOG
WHERE BIZ_EVENT_ID = <event_id>;
```

BIN_STATUS will be one of:

Status	Meaning
COMPLETE	The file has been fully ingested into the output tables.
PENDING	Ingestion is still in progress.
ERROR	Ingestion failed. Check BIZ_MSG for details.

Phase 1: Generate Seed EDI Files (EDl_FROM_BIN)

The first step is to generate sample EDI files from SERENEDI's built-in sample data. This uses the manual event submission pattern — you INSERT rows directly into BIZ_EVENT, referencing the USP_*_EXTRACT procedures that query the SAMPL_* tables.

```
-- 837P - Professional Claim
INSERT INTO SERENEDI.MAIN.BIZ_EVENT
(BIZ_EVENT_ID, BIZ_TRIGGER_ID, EVENT_DATE,
EVENT_CRIT, SOURCE_NM, EVENT_DATA1,
EVENT_DATA2, EVENT_DATA3)
SELECT SERENEDI.MAIN.BIZ_EVENT_SEQ.NEXTVAL,
8, CURRENT_TIMESTAMP(), NULL, 'USER',
'CALL SERENEDI.MAIN.USP_837P_EXTRACT()',
'@SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt',
NULL;

-- 835 - Claim Payment/Remittance
INSERT INTO SERENEDI.MAIN.BIZ_EVENT
(BIZ_EVENT_ID, BIZ_TRIGGER_ID, EVENT_DATE,
EVENT_CRIT, SOURCE_NM, EVENT_DATA1,
```

```

EVENT_DATA2, EVENT_DATA3)
SELECT SERENEDI.MAIN.BIZ_EVENT_SEQ.NEXTVAL,
8, CURRENT_TIMESTAMP(), NULL, 'USER',
'CALL SERENEDI.MAIN.USP_835_EXTRACT()',
'@SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_835.txt',
NULL;

```

```

-- 270 - Eligibility Inquiry
INSERT INTO SERENEDI.MAIN.BIZ_EVENT
(BIZ_EVENT_ID, BIZ_TRIGGER_ID, EVENT_DATE,
EVENT_CRIT, SOURCE_NM, EVENT_DATA1,
EVENT_DATA2, EVENT_DATA3)
SELECT SERENEDI.MAIN.BIZ_EVENT_SEQ.NEXTVAL,
8, CURRENT_TIMESTAMP(), NULL, 'USER',
'CALL SERENEDI.MAIN.USP_270_EXTRACT()',
'@SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt',
NULL;

```

Now kick off processing:

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: You submitted three events to the BIZ_EVENT work queue. Each event tells the engine to call a specific extract procedure (EVENT_DATA1), which queries the sample data tables and produces a result set. The engine then encodes that result set into an EDI file and writes it to the stage path specified in EVENT_DATA2.

Verify: Confirm that all three events completed successfully:

```

SELECT BIZ_EVENT_ID, EVENT_CRIT, SUMMARY
FROM SERENEDI.MAIN.BIZ_EVENT
ORDER BY EVENT_DATE DESC
LIMIT 10;

```

All three rows should show a SUMMARY of SUCCESS. You can also confirm the files were created:

```
LIST @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/;
```

You should see TEST_837P.txt, TEST_835.txt, and TEST_270.txt.

Phase 2: Compliance Checking (INTEGRITY)

Now let's run SERENEDI's compliance rules engine against the EDI files we just generated. This uses the automatic event submission pattern — you copy files into a watched subfolder and let TRIGGER_SCAN generate the events.

```

COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/INTEGRITY/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/INTEGRITY/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_835.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/INTEGRITY/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt;

```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: You copied three EDI files into the EDI_IN/INTEGRITY/ subfolder. When HEARTBEAT called TRIGGER_SCAN, it detected the new files, matched them against the INTEGRITY trigger in BIZ_TRIGGER, and generated three BIZ_EVENT rows — one per file. The engine then ran its compliance rules against each file and produced HTML integrity reports.

Verify: Check that the events completed:

```

SELECT BIZ_EVENT_ID, EVENT_CRIT, SUMMARY
FROM SERENEDI.MAIN.BIZ_EVENT
WHERE BIZ_TRIGGER_ID != 8

```

```
ORDER BY EVENT_DATE DESC
LIMIT 10;
```

The output reports are HTML files written to the ERR stage under the INTEGRITY subfolder:

```
LIST @SERENEDI_DATA.STAGES.ERR/INTEGRITY/;
```

By default, the reports generated through the trigger pipeline include both the compliance errors and the original EDI segments, so you can see exactly which segments triggered each rule. If you want error-only reports without the segment detail, you can submit INTEGRITY events manually with EVENT_DATA3 set to MSG_ONLY.

Phase 3: Decode EDI to CSV (CSV_FROM_EDI)

```
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/CSV_FROM_EDI/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/CSV_FROM_EDI/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_835.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/CSV_FROM_EDI/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt;
```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: SERENEDI decoded each EDI file and produced a CSV representation containing every data element from the original transaction, using SERENEDI's predefined column-naming convention. These CSV files are written to the CSV_OUT stage.

Verify:

```
LIST @SERENEDI_DATA.STAGES.CSV_OUT/CSV_FROM_EDI/;
```

You should see TEST_837P.txt.csv, TEST_835.txt.csv, and TEST_270.txt.csv.

Phase 4: Decode EDI to XML (XML_FROM_EDI)

```
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/XML_FROM_EDI/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/XML_FROM_EDI/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_835.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/XML_FROM_EDI/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt;
```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: Same as CSV_FROM_EDI, but the output is XML. Each EDI file's full content is represented in a structured XML document.

Verify:

```
LIST @SERENEDI_DATA.STAGES.XML_OUT/XML_FROM_EDI/;
```

You should see TEST_837P.txt.xml, TEST_835.txt.xml, and TEST_270.txt.xml.

Phase 5: Decode EDI to Flat BIN Tables (EDI_TO_BIN)

```
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_BIN/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_BIN/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_835.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_BIN/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt;
```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: SERENEDI decoded each EDI file and loaded the data into denormalized Flat BIN tables in the SERENEDI_DATA.OUTPUT schema. Each transaction set gets its own table (e.g., BIN_5010_837P, BIN_5010_835). If a table for that transaction set already exists, new columns are added automatically as needed.

Verify: Remember that for BIN ingestion, BIZ_EVENT reaching SUCCESS means the file was accepted — check BIN_LOG to confirm the data is fully loaded:

```
SELECT BIN_ID, BIN_FILENAME, BIN_STATUS
FROM SERENEDI.MAIN.BIN_LOG
ORDER BY BIN_ID DESC
LIMIT 10;
```

Once BIN_STATUS shows COMPLETE, you can query the output tables directly:

```
SELECT * FROM SERENEDI_DATA.OUTPUT.BIN_5010_837P LIMIT 10;
SELECT * FROM SERENEDI_DATA.OUTPUT.BIN_5010_835 LIMIT 10;
SELECT * FROM SERENEDI_DATA.OUTPUT.BIN_5010_270 LIMIT 10;
```

This is where the power of the Flat BIN system becomes apparent. All the data from the EDI file is now in a standard Snowflake table, queryable with SQL. For example, to get all unique subscriber last names from the 837P data:

```
SELECT DISTINCT L2010BA_NM103_PERSN_LNM
FROM SERENEDI_DATA.OUTPUT.BIN_5010_837P;
```

Phase 6: Decode EDI to Hierarchical Database (EDI_TO_HDB)

```
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_HDB/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_HDB/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_835.txt;
COPY FILES INTO @SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_HDB/
FROM @SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt;
```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: SERENEDI decoded each EDI file into the Hierarchical Database format. Unlike Flat BIN, HDB stores each EDI loop in its own table, preserving the parent-child relationships from the original transaction. This eliminates data redundancy but requires you to know how the tables relate to each other. To help with this, the HDB tables include parent-child key constraints that provide hints about these relationships. A complete reference of the HDB table relationships for each transaction set is provided in the appendix of this manual.

Verify: Check BIN_LOG for HDB entries (BIN_TYPE = 103):

```
SELECT BIN_ID, BIN_FILENAME, BIN_STATUS
FROM SERENEDI.MAIN.BIN_LOG
WHERE BIN_TYPE = 103
ORDER BY BIN_ID DESC
LIMIT 10;
```

Once complete, you can explore the HDB tables in SERENEDI_DATA.OUTPUT. Each transaction set will have multiple tables, one per loop level.

Phase 7: Round-Trip — CSV to EDI (CSV_TO_EDI)

Now take the CSV files generated in Phase 3 and encode them back into EDI:

```
COPY FILES INTO @SERENEDI_DATA.STAGES.CSV_IN/CSV_TO_EDI/
FROM @SERENEDI_DATA.STAGES.CSV_OUT/CSV_FROM_EDI/TEST_837P.txt.csv;
COPY FILES INTO @SERENEDI_DATA.STAGES.CSV_IN/CSV_TO_EDI/
FROM @SERENEDI_DATA.STAGES.CSV_OUT/CSV_FROM_EDI/TEST_835.txt.csv;
```

```
COPY FILES INTO @SERENEDI_DATA.STAGES.CSV_IN/CSV_TO_EDI/  
FROM @SERENEDI_DATA.STAGES.CSV_OUT/CSV_FROM_EDI/TEST_270.txt.csv;
```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

What just happened: SERENEDI read each CSV file, interpreted the column names using its built-in CGIF2 mappings, and encoded the data back into valid EDI format. The resulting EDI files are written to EDI_OUT.

Verify:

```
LIST @SERENEDI_DATA.STAGES.EDI_OUT/CSV_TO_EDI/;
```

Because SERENEDI's transformations are bidirectional and lossless for HIPAA-compliant files, the EDI files produced here will be a binary-accurate representation of the originals. This round-trip capability is one of SERENEDI's key features — it means you can decode an EDI file, inspect or modify the data in CSV form, and re-encode it without data loss.

Phase 8: Round-Trip — XML to EDI (XML_TO_EDI)

The same round-trip works with XML:

```
COPY FILES INTO @SERENEDI_DATA.STAGES.XML_IN/XML_TO_EDI/  
FROM @SERENEDI_DATA.STAGES.XML_OUT/XML_FROM_EDI/TEST_837P.txt.xml;  
COPY FILES INTO @SERENEDI_DATA.STAGES.XML_IN/XML_TO_EDI/  
FROM @SERENEDI_DATA.STAGES.XML_OUT/XML_FROM_EDI/TEST_835.txt.xml;  
COPY FILES INTO @SERENEDI_DATA.STAGES.XML_IN/XML_TO_EDI/  
FROM @SERENEDI_DATA.STAGES.XML_OUT/XML_FROM_EDI/TEST_270.txt.xml;
```

```
CALL SERENEDI.MAIN.HEARTBEAT();
```

Verify:

```
LIST @SERENEDI_DATA.STAGES.EDI_OUT/XML_TO_EDI/;
```

What You've Accomplished

In this walkthrough, you have:

- Generated seed EDI files for three transaction sets from SERENEDI's built-in sample data (EDI_FROM_BIN).
- Run HIPAA compliance checking and reviewed HTML integrity reports (INTEGRITY).
- Decoded EDI files into CSV, XML, Flat BIN tables, and Hierarchical Database tables.
- Queried decoded data directly in Snowflake using standard SQL.
- Round-tripped CSV and XML back to EDI, producing binary-accurate reproductions of the originals.

Every workflow demonstrated here applies identically to all 13 supported 5010 transaction sets. To run the full suite, simply repeat the same patterns with the remaining seed file generators (USP_271_EXTRACT through USP_834_EXTRACT).

Where to Go from Here

The walkthrough covered the core transformation and analysis workflows. Additional capabilities documented later in this manual include:

Encoding from BIN and HDB — Generating EDI files from data already loaded into Flat BIN or Hierarchical Database tables. This uses the BIN_LOG to identify which BIN items to encode.

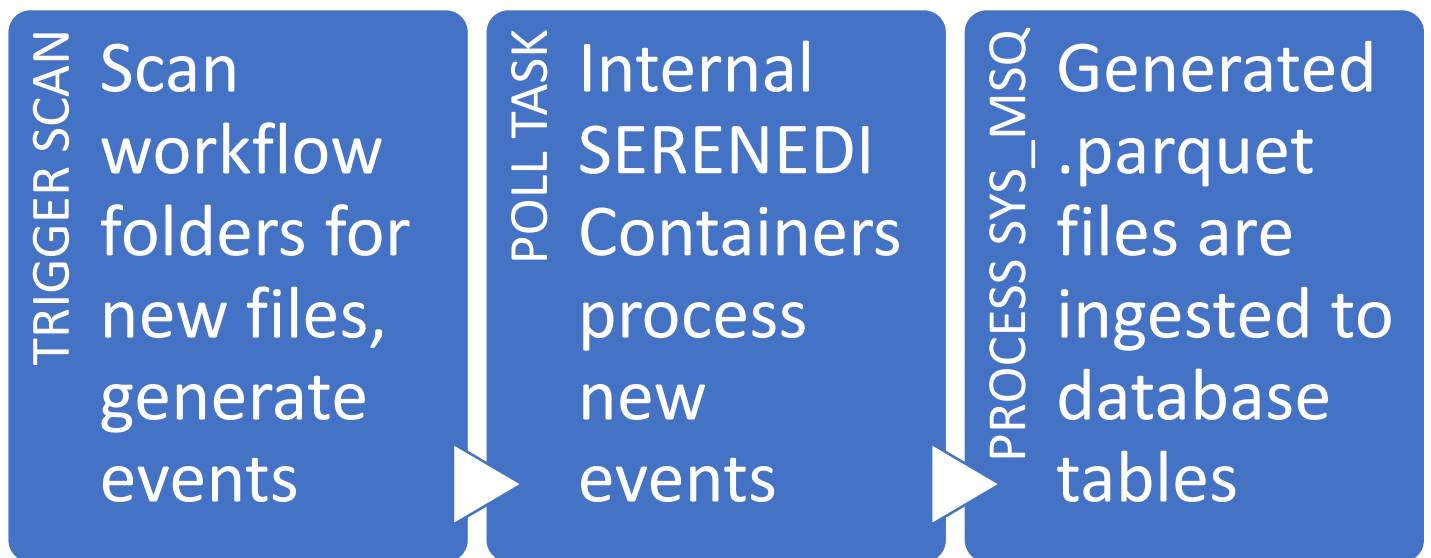
HEARTBEAT_POLL for production — Configuring SERENEDI to run unattended with scheduled polling.

The CGIF2 mapping convention — Understanding the column-naming system that drives SERENEDI’s bidirectional transformations.

Building your own data extracts — Creating stored procedures that produce data in SERENEDI’s expected format, enabling you to encode your enterprise data into EDI.

System Architecture

SERENEDI’s automation system revolves around a call to a stored procedure called SERENEDI.MAIN.USP_HEARTBEAT. It is a unified orchestrator that can be run either ad-hoc or as an ongoing job service. When passed a single integer argument, it will set up the ongoing job service and execute the heartbeat every *X* minutes. When a 0 is passed, the background job is terminated. Regardless of how it is called, each heartbeat call does the following actions:



USP_HEARTBEAT

TRIGGER_SCAN – Scan the workflow folders and generate new events

HEARTBEAT_POLL – Execute SERENEDI Containers until new events have been processed

PROCESS_SYS_MSQ – Process .parquet file ingestion to SERENEDI_DATA.OUTPUT schema.

PROCESS_DST_TBL – Creates new SERENEDI database tables and schema hints

PROCESS_DST_INSERT – Bulk inserts the data from the .parquet files

UPDATE_BIN_STATUS – Updates the BIN_LOG table to indicate what files were successfully processed

Trigger Scan

The Trigger Scan pass will examine the workflow file paths defined in the BIZ_TRIGGER table and shown in the table to the right and attempt to move a file from the *Initial* folder to the *Source* folder. If the file does not exist in the Source folder, it is successfully moved and the TRIGGER SCAN procedure generates an event, on the file that now exists in the Source folder (shown in green).

This is a simple way to prevent double processing a file: files that have the same name as one already processed will simply be stuck in the Initial folder.

Once the file has been moved, the Trigger Scan phase generates a new event for each file moved. Two workflows cannot be triggered by the Trigger Scan mechanism and must be generated manually; they are EDI_FROM_BIN and GENERATE_999.

Poll Task

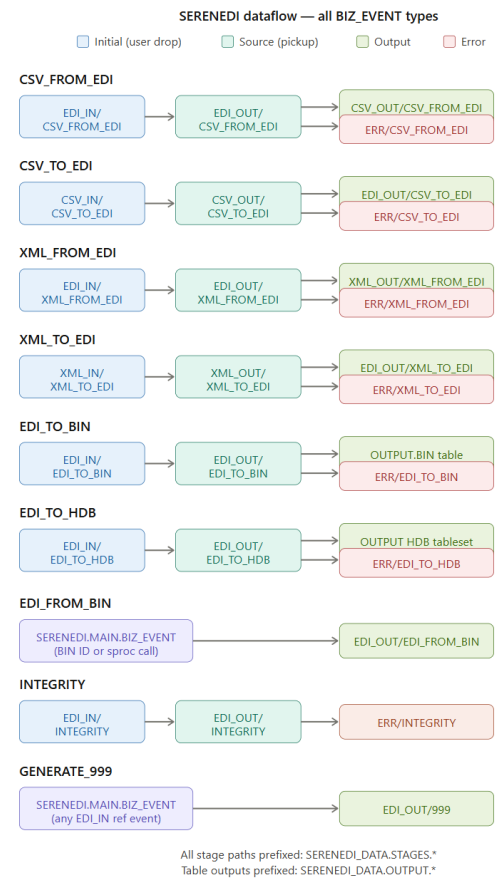
When all of the workflow folders have been analyzed by the Trigger Scan phase, SERENEDI will call HEARTBEAT_POLL, which will launch up to two Snowpark Container Services (SPCS) containers and process the new events in an indeterminate order. For the EDI to CSV and XML workflows, the container will ingest the EDI file, convert it into SERENEDI mapped equivalents, and write the generated artefact to the output folder (shown in Green in the previous diagram). If there were any “soft” integrity errors, they are logged to the BIZ_MSG table. If there were “hard” integrity errors that prevented parsing, then not only are these errors logged to the BIZ_MSG table, but the SUMMARY field of the BIZ_EVENT row for this event is labeled “CRITICAL ERROR” instead of the normal “SUCCESS” and the *Source* file is placed in the *Error* folder.

For CSV / XML to EDI workflows, it simply works in reverse: the CSV or XML file is consumed and, so long as there weren’t critical errors, the EDI file is written to the output folder. If the encoding process encountered a critical error, the original file is moved to the Error folder, just like before.

For EDI_TO_BIN, the file is converted into a single .parquet file that will be processed in the next phase. For EDI_TO_HDB, a single incoming EDI file is converted to potentially many .parquet files, with one output file present for every distinct loop in the original EDI file. These .parquet files are stored temporarily to the SERENEDI.RUNTIME.TMP stage. Furthermore, all parquet files are logged to the SYS_MSQ table. In both cases, the original EDI file is entered into the BIN_LOG table and assigned to a unique BIN_ID. In the finalized output tables, this BIN_ID is associated with the data belonging to that file.

Process SYS_MSQ

This phase follows once all outstanding events have been processed. When SERENEDI completes an event that involves ingesting files into SERENEDI’s BIN system, SERENEDI will mark the event as SUCCESS but the BIN_LOG entry will be marked as PENDING. This BIN_LOG will be associated with at least one or more .parquet files logged within the SYS_MSQ table. When this phase starts to process .parquet files, it calls PROCESS_DST_TBL(<destination table>) which checks to see if the



This outputs a single CGIF2 Flat formatted CSV file from a given EDI file. Unlike in the BIN/HDB data projections, all of the data is presented as quote-delimited text fields. This file can be used in several ways:

- Import the data into text-only database tables
- Learn SERENEDI's CSV mapping schema

Note that the columns will always be *dynamic* – SERENEDI never outputs a single set of maps. It will map the columns exactly according to the data within the original file, so the number of columns will be variable. In all cases, however, the first column will contain a two character prefix denoting the transaction, and a last column of NEWROW to enable encoding of certain types of repeating segments.

CSV_TO_EDI	
Initial Folder	Source Folder
SERENEDI_DATA.STAGES.CSV_IN/CSV_TO_EDI	SERENEDI_DATA.STAGES.CSV_OUT/CSV_TO_EDI
Output Folder	Error Folder
SERENEDI_DATA.STAGES.EDI_OUT/CSV_TO_EDI	SERENEDI_DATA.STAGES.ERR/CSV_TO_EDI

This is the mirror to the previous workflow, and provides an interface for SERENEDI to generate an EDI file from a SERENEDI-formatted CSV file. The primary business purpose of the CSV to EDI schema is for *legacy enterprise interfaces*. Older healthcare enterprise systems that may lack XML or database connectivity often have well-documented functions to generate outbound CSV files.

XML_FROM_EDI	
Initial Folder	Source Folder
SERENEDI_DATA.STAGES.EDI_IN/XML_FROM_EDI	SERENEDI_DATA.STAGES.EDI_OUT/XML_FROM_EDI
Output Folder	Error Folder
SERENEDI_DATA.STAGES.XML_OUT/XML_FROM_EDI	SERENEDI_DATA.STAGES.ERR/XML_FROM_EDI

This interface translates an EDI file directly into CGIF2-formatted XML files. These XML files can be consumed by a wide variety of interfaces: custom APIs, XML-oriented data storage, or other applications. Furthermore, it can also be used to learn how to map specific situations into XML to support development for the following XML_TO_EDI workflow.

XML_TO_EDI	
Initial Folder	Source Folder
SERENEDI_DATA.STAGES.XML_IN/XML_TO_EDI	SERENEDI_DATA.STAGES.XML_OUT/XML_TO_EDI
Output Folder	Error Folder
SERENEDI_DATA.STAGES.EDI_OUT/XML_TO_EDI	SERENEDI_DATA.STAGES.ERR/XML_TO_EDI

This mirrors the previous interface, and represents a powerful, straightforward way to generate complex EDI transactions by first encoding the data in CGIF2-XML format and then sending it to this workflow.

In the following two examples, we can see three segments from an EDI file and it's exact equivalent in CGIF2 XML:

```
0000048 L2010CA DMG|D8|19840401|F
0000049 L2300 CLM|PAT00036161|502|||13>B>1|Y|A|Y|Y
0000050 L2300 REF|D9|CLM0000000250
```

```
0000169 <DMG02_D8>19840401</DMG02_D8>
0000170 <DMG03_PAT_GNDR_CD>F</DMG03_PAT_GNDR_CD>
0000171 </L2010CA>
0000172 </L2000C>
0000173 <L2300>
0000174 <CLM01_PT_CTL_NR>PAT00036161</CLM01_PT_CTL_NR>
0000175 <CLM02_TOT_CLM_CHG_AMT>502</CLM02_TOT_CLM_CHG_AMT>
0000176 <CLM0501_POS_CD>13</CLM0501_POS_CD>
0000177 <CLM0503_CLM_FREQ_CD>1</CLM0503_CLM_FREQ_CD>
0000178 <CLM06_PVD_SUPP_SIG_IND>Y</CLM06_PVD_SUPP_SIG_IND>
0000179 <CLM07_PLAN_PART_CD>A</CLM07_PLAN_PART_CD>
0000180 <CLM08_BEN_ASGT_CRT_IND>Y</CLM08_BEN_ASGT_CRT_IND>
0000181 <CLM09_RELS_NFO_CD>Y</CLM09_RELS_NFO_CD>
0000182 <REF_CLM_NR>CLM0000000250</REF_CLM_NR>
```

The first example will be human understandable to very few people – the second example, on the other, is pretty easy to follow. Patient Control Number, Total Claim Charge Amount, Place of Service Code, and so on. The EDI elements REF, D9, and CLM0000000250 are all encapsulated in the much more comprehensible XML map, <REF_CLM_NR>CLM0000000250</REF_CLM_NR>.

EDI_TO_BIN	
Initial Folder	Source Folder
SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_BIN	SERENEDI_DATA.STAGES.EDI_OUT/EDI_TO_BIN
Output	Error Folder
SERENEDI_DATA.OUTPUT	SERENEDI_DATA.STAGES.ERR/EDI_TO_BIN

This workflow 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. Note that even when the event is completed, there will still be a delay before the data is actually ingested; the BIN_LOG will have the actual status of the data.

The name of the BIN will follow the naming schemes BIN_5010_<transaction set number>.

This workflow does not have a file output; it will send the data directly to a single, strongly-typed database table. This interface is primarily for *EDI editing*. Because of the extremely denormalized nature of the data, it's not really ideal for querying or data warehousing. These same attributes make it ideal for another role: *editing and re-exporting*. There are situations when Federal and State Regulatory bodies need your enterprise's transactions bundled together, given different headers and shipped off as a new and different EDI file. Here, the denormalized nature of the data makes this an ideal format for modification and re-transmission.

EDI_TO_HDB	
Initial Folder	Source Folder
SERENEDI_DATA.STAGES.EDI_IN/EDI_TO_HDB	SERENEDI_DATA.STAGES.EDI_OUT/EDI_TO_HDB
Output	Error Folder
SERENEDI_DATA.OUTPUT	SERENEDI_DATA.STAGES.ERR/EDI_TO_HDB

This workflow ingests files into the Hierarchical HDB BIN system built into SERENEDI. The HDB stores data hierarchically, with one table present for each loop and all tables joined by keys. Like the previous workflow, if the tables do not exist, they will be created, and new columns will also be dynamically created as well.

The default table names are <specification 3-letter prefix>_<loop short name>. The specification prefix can be found in the Appendix as well as the table within the Hierarchical Database section.

For example, the seed 834 file will decode to the following tables: BEN_ISA, BEN_GSHDR, BEN_STHDR, BEN_L1000A, BEN_L1000B, BEN_L2000, BEN_L2100A, BEN_L2300.

EDI_FROM_BIN	
Initial Folder	Source Folder
BIZ_EVENT.EVENT_DATA1 = Stored procedure call OR BIN_ID of a previously ingested file BIZ_EVENT.EVENT_DATA2 = blank, a file name and extension, or a fully-pathed destination file	
Output	Error Folder
SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN	

This workflow generates EDI files from the database artefacts generated from the BIN/HDB system, *or* from user-generated stored procedures.

Events for this workflow can *only* come from users making inserts into the BIZ_EVENT table. EDI_FROM_BIN is used for three purposes: regenerate an EDI from a previously loaded BIN record, regenerate an EDI from a previously loaded HDB record, or generate an EDI directly from the stored procedure.

Here is a repeat of the example provided in the Functional Walkthrough:

```
INSERT INTO SERENEDI.MAIN.BIZ_EVENT
  (BIZ_EVENT_ID, BIZ_TRIGGER_ID, EVENT_DATE,
   EVENT_CRIT, SOURCE_NM, EVENT_DATA1,
   EVENT_DATA2)
SELECT SERENEDI.MAIN.BIZ_EVENT_SEQ.NEXTVAL,
  8, CURRENT_TIMESTAMP(), NULL, 'USER',
  'CALL SERENEDI.MAIN.USP_837P_EXTRACT()',
  '@SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_837P.txt';
```

In this example, EVENT_DATA1 is used to convey the exact stored procedure path for a call to extract an 837P from the Sample Data; this procedure is shipped with the SERENEDI environment. If this argument is populated with a BIN_ID, whether encoded with the BIN or HDB systems, it will use that as the data source for encoding the file. For these two cases, EVENT_DATA2 will be defaulted to the original filename when the file was imported. Otherwise, users can put in just a filename and it will appear in the Output default location, or alternatively they can specify a fully qualified destination

path for the EDI file to create. It is the responsibility of the user to ensure the SERENEDI has the correct privileges granted to it into order to create a file at that location.

INTEGRITY	
Initial Folder	Source Folder
SERENEDI_DATA.STAGES.EDI_IN/INTEGRITY	SERENEDI_DATA.STAGES.EDI_OUT/INTEGRITY
Output	Error Folder
SERENEDI_DATA.STAGES.ERR/INTEGRITY	

This workflow will show the output of the SERENEDI engine's integrity analysis on an EDI file. All of the supported specifications cover SNIP Type 1 and 2:

Type 1: EDI envelope structures, segment counts

Type 2: Correct segment order, required elements, valid data types, min/max lengths

The 270, 271, 834, 835, 837I and 837P have three further types of testing:

Type 3: Balance checks

Type 4: Inter-segment situational requirements

Type 5: External Code set validation

In this example, the sample 835 file generated from SERENEDI.MAIN.USP_835_EXTRACT() has been modified from the original so that it has intentionally bad balancing. When this example is fed through the Integrity workflow, the result on the right is generated as an HTML file:

```

CLP*PAT0012983*1*800.8*401.4*12.12*HM*CLM000000010*13~> 3000004$THDR $THDR_240:Total Provider Payment should equal the sum of all Claim Payments minus any possible Provider Adjustments
NMI*QC*1*BREAD*SHANNON****MI*SUB000000093~> 3000005$THDR TRN|1|CHK00001|150001342
REF*1L*POLCY0001~> 3000006L1000AN1|PR|LOWEST BIDDER HMO
REF*CE*C001~> 3000007L1000AN3|400 FICTION ST
DTM*232*20180830~> 3000008L1000AN4|SAN FRANCISCO|CA|94117
DTM*233*20180902~> 3000009L1000AREF|ZU|150001342
AMT*AU*800.8~> 3000010L1000APER|BL|TECH CONTACT|TE|4155550001
SVC*HC>99202>RT*129.4*64.85**1~> 3000011L1000N1|PE|CHEAPEST EVER HMO|XX|1234567893
DTM*472*20180831~> 3000012L1000N3|200 FICTION ST
CAS*CO*45*64.85~> 3000013L1000N4|SAN FRANCISCO|CA|94117
AMT*B6*129.7~> 3000014L1000BREF|TJ|150000676
SVC*HC>99202>RT*86.14*43.07**1~> 3000015L2000 LX1
DTM*472*20180831~> 3000016L2100 CLP|PAT0012983|1|800.8|401.4|12.12|HM|CLM000000010|13
CAS*PR*2*8.62~> 3000016L2100 L2100_150:Claim Payment Amount should equal Claim Charge Amount minus all claim and service adjustments
CAS*CO*45*34.41~> 3000016L2100 L2100_155:Claim Patient Responsibility Amount should equal the sum of all Patient Responsibility adjustments
AMT*B6*86.14~> 3000017L2100 NMI|QC|1|BREAD|SHANNON|MI|SUB000000093
SVC*HC>99202>RT*90.3*45.15**1~> 3000018L2100 REF|1L|POLCY0001
DTM*472*20180901~> 3000019L2100 REF|CE|C001
CAS*CO*45*45.15~> 3000020L2100 DTM|232|20180830
AMT*B6*90.3~> 3000021L2100 DTM|233|20180902
SVC*HC>99202>494.66*247.33**1~> 3000022L2100 AMT|AU|800.8
DTM*472*20180902~> 3000023L2110 SVC|HC>99202|129.4|64.85||1
CAS*PR*3*2.5~> 3000023L2110 L2110_105:Line Item Paid Amount must equal Line Item Charge Amount minus the sum of all line adjustments
CAS*CO*45*244.83~> 3000024L2110 DTM|472|20180830
AMT*B6*494.66~> 3000025L2110X CAS|CO|45|64.85
SVC|HC>99202>RT|86.14|43.07||1 3000026L2110 AMT|86|129.7
L2110_105:Line Item Paid Amount must equal Line Item Charge Amount minus the sum of all line adjustments 3000027L2110

```

If the integrity event is user generated and EVENT_DATA3 is set to MSG_ONLY, then the HTML will contain *only* the integrity messages and omit the original segments.

GENERATE 999	
Initial Folder	Source Folder
BIZ_EVENT.EVENT_DATA1 = BIZ_EVENT_ID of incoming EDI file	
Output	Error Folder
SERENEDI_DATA.STAGES.EDI_OUT/999	

The Generate 999 is oriented to fulfill a common responsibility: acknowledge an incoming EDI file received from a trading partner. SERENEDI has a basic mechanism for this purpose. If the file can be parsed, even with many syntax errors, it will label the incoming transaction sets as *Accepted* within the outgoing 999 Acknowledgement transaction. If, however, there was a critical error that prevented parsing of this incoming file, then the outgoing 999 is going to label the specific segment where it stopped parsing as a critical error, and all transactions of that file will be labeled as *Rejected*.

This workflow requires that EVENT_DATA1 column be populated with another BIZ_EVENT_ID that handles an incoming EDI file. Depending on the original workflow, it will decode the file associated with that event and generate a 999 based on the results.

If there are requirements for a more detailed 999 response, or a simpler response such as a ISA/TA1/IEA acknowledgment, the 999 is supported as a normal transaction for generation and encoding.

TECHNICAL REFERENCE

Supported Transaction Set Identifiers

5010 Transaction Set Name	Two-digit First Column Prefix	Three-Digit Table Name Prefix
270 Health Care Eligibility Benefit Inquiry	M0, M1	BNQ
271 Health Care Eligibility Benefit Response	N0, N1	BNR
276 Health Care Claim Status Request	O0	HCQ
277 Health Care Claim Status Response	P0	HCR
277 CA Health Care Claim Acknowledgment	P5	HCK
278 REQ Health Care Services Review – Request for Review	Q0	HVQ
278 RESP Health Care Services Review – Response	R0	HVR
278 NOT Health Care Services Review Information - Notification	Q1	SVN
278 ACK Health Care Services Review Information - Acknowledgement	R1	SVK
820 Payroll Deducted and Other Group Premium Payment for Insurance Products	S0	PRL
820X Health Insurance Exchange Related Payments	S5	HIX
824 Application Reporting for Insurance	P7	APP
834 Benefit Enrollment and Maintenance	T0, T1	BEN
835 Health Care Claim Payment/Advice	U0, U1	PMT
837 Health Care Claim: Dental	V0, V1, V2	CMD
837 Health Care Claim: Institutional	W0, W1, W2	CMI
837 Health Care Claim: Professional	X0, X1	CMP
999 Acknowledgment	Z1	ACK

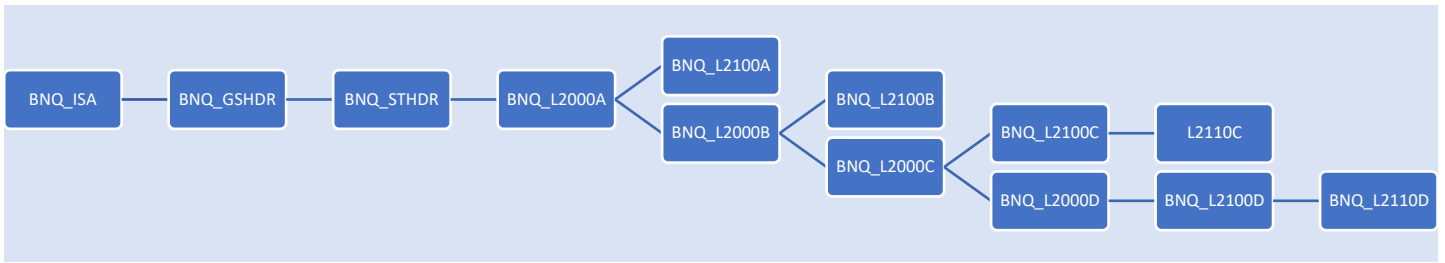
SCHEMA REFERENCE

These diagrams show the hierarchical relationship between the loops within a transaction. Each transaction has a unique prefix, and the full name of the table as it would appear in the Snowflake schema is shown here.

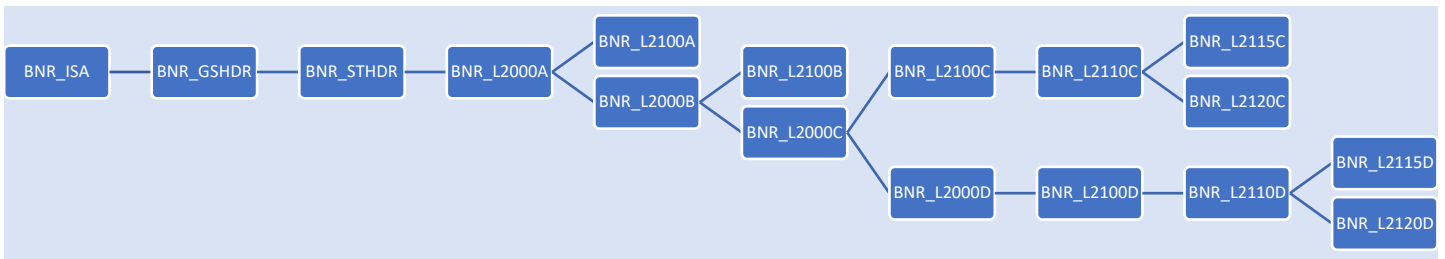
Any loop that has a name ending in X or Y is a *cutout* loop. These loops are not defined in the HIPAA Implementation Guides – instead, they are a SERENEDI convention in which a segment defined within the HIGs as having unlimited repeats is artificially established as another loop. By isolating this segment information, SERENEDI can make it easier to manage the rest of the information for encoding and decoding.

These tables *only* get created when an EDI file is parsed and decoded to the HDB BIN system; they are not manually generated. Also note that each of these definitions relate to the *originally published* transaction without Addenda; there may be additional loops defined within the Addenda A1 and Addenda A2 as described within the X12-published Addenda.

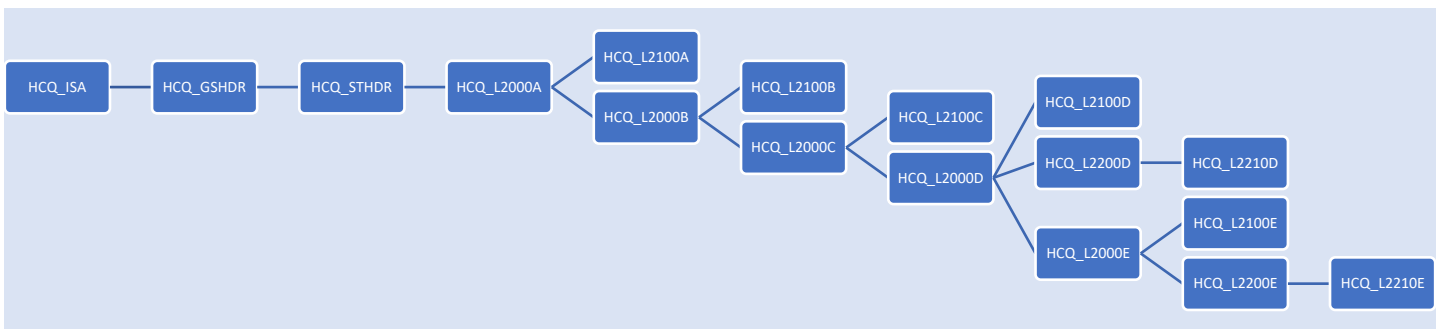
5010_270 / M0 Health Care Eligibility Benefit Inquiry / BNQ



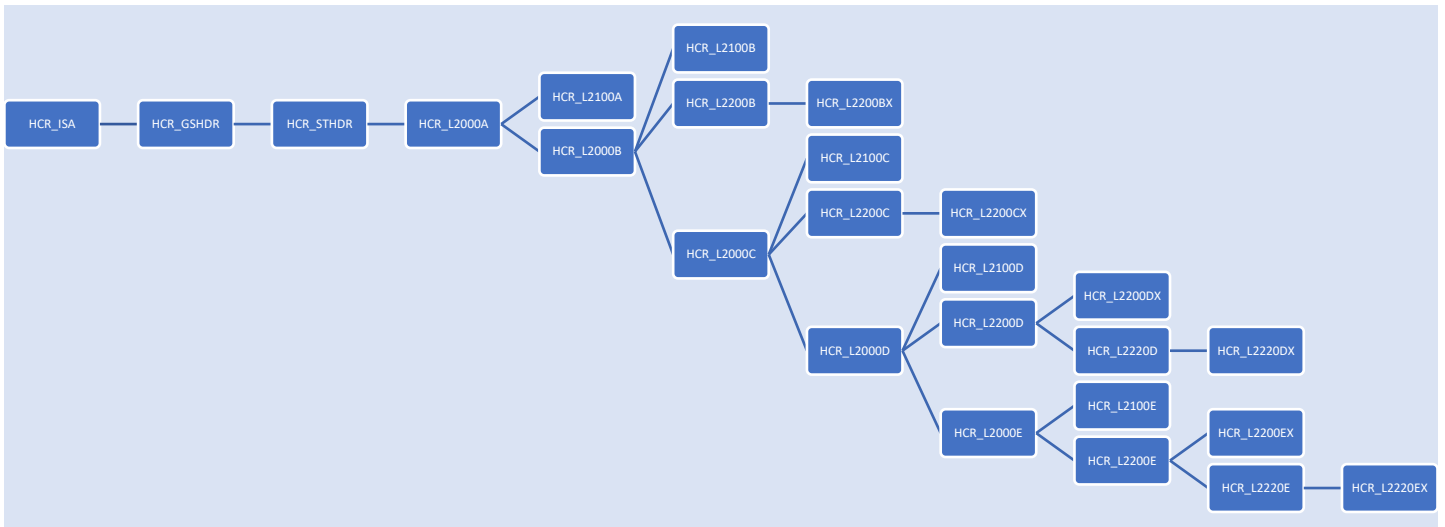
5010_271 / N0 Health Care Eligibility Benefit Response / BNR



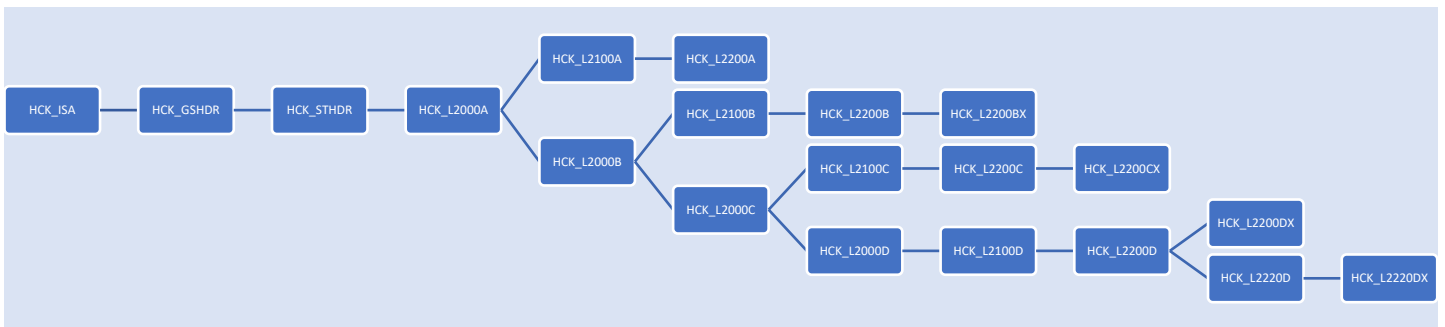
5010_276 / 00 Health Care Claim Status Request / HCQ



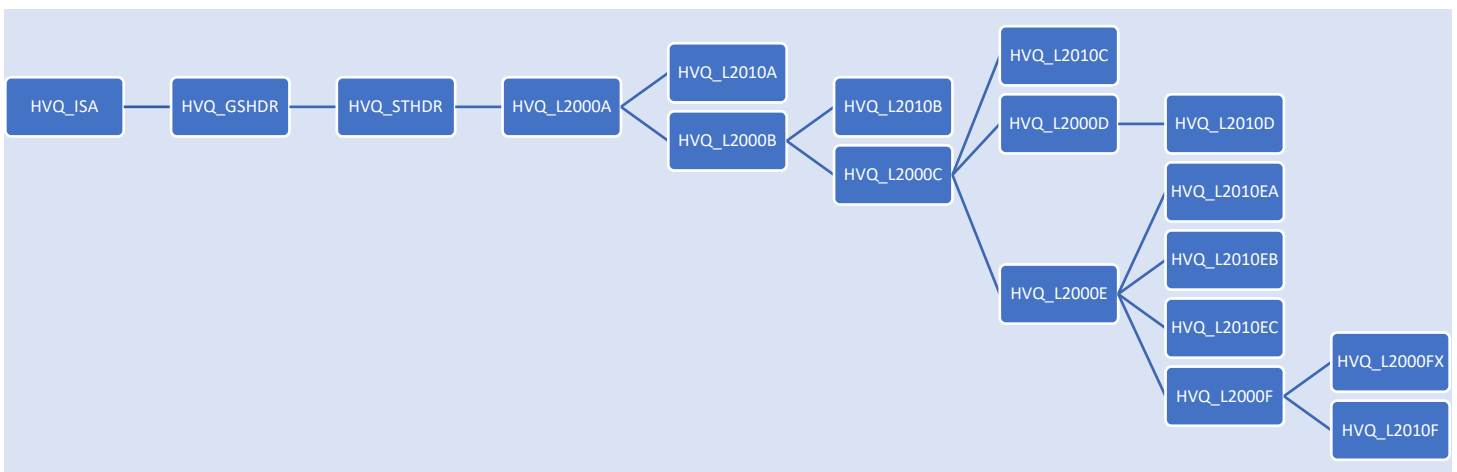
5010_277 / P0 Health Care Claim Status Response / HCR



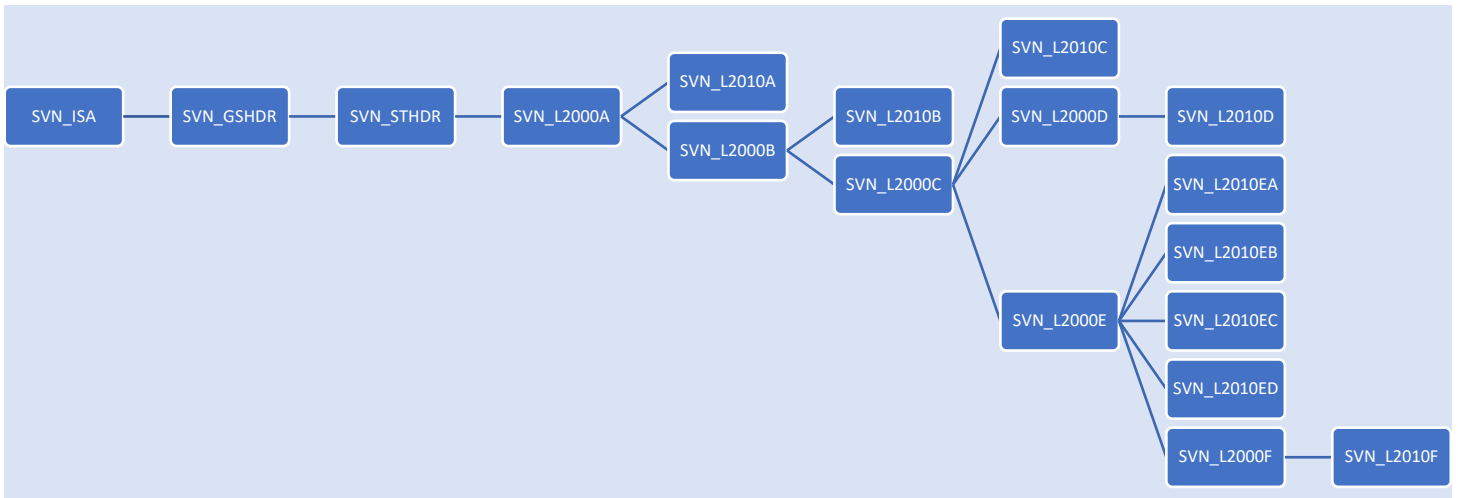
5010_277CA / P5 Health Care Claim Acknowledgment / HCK



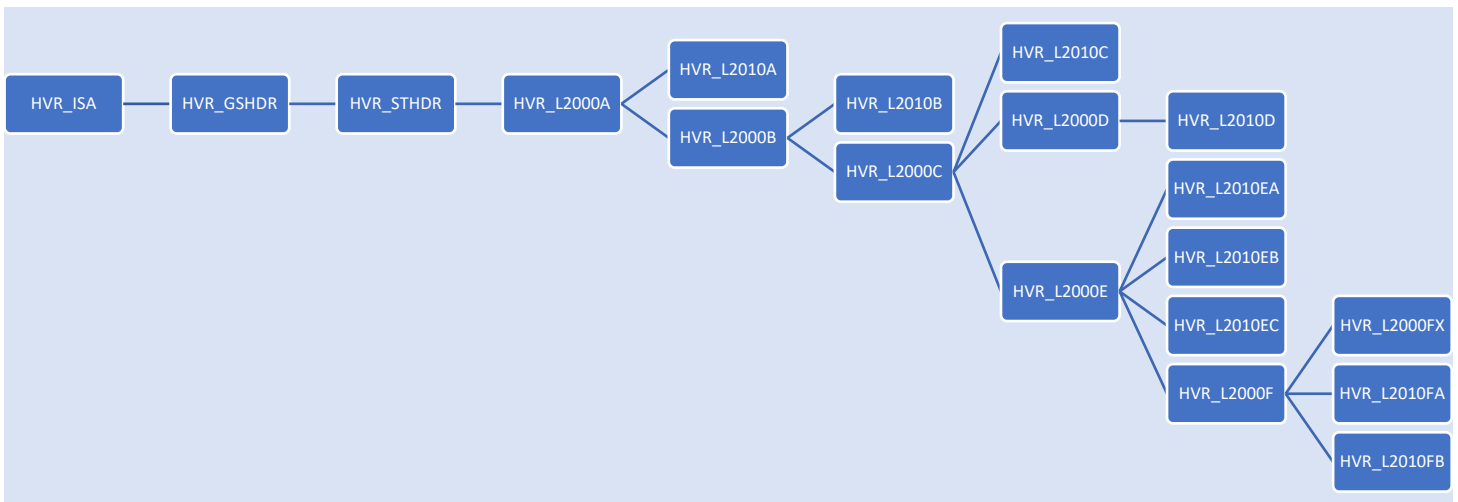
5010_278_REQ / Q0 Health Care Services Review - Request for Review / HVQ



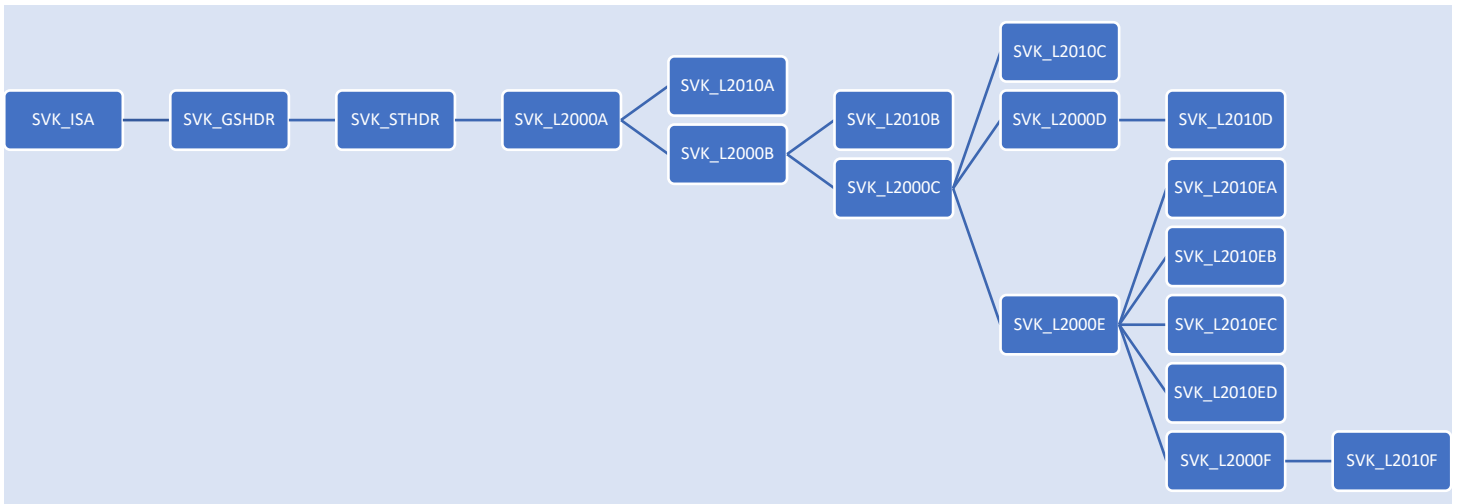
5010_278_NOT / Q1 Health Care Services Review - Notification / SVN



5010_278_RESP / R0 Health Care Services Review - Response / HVR

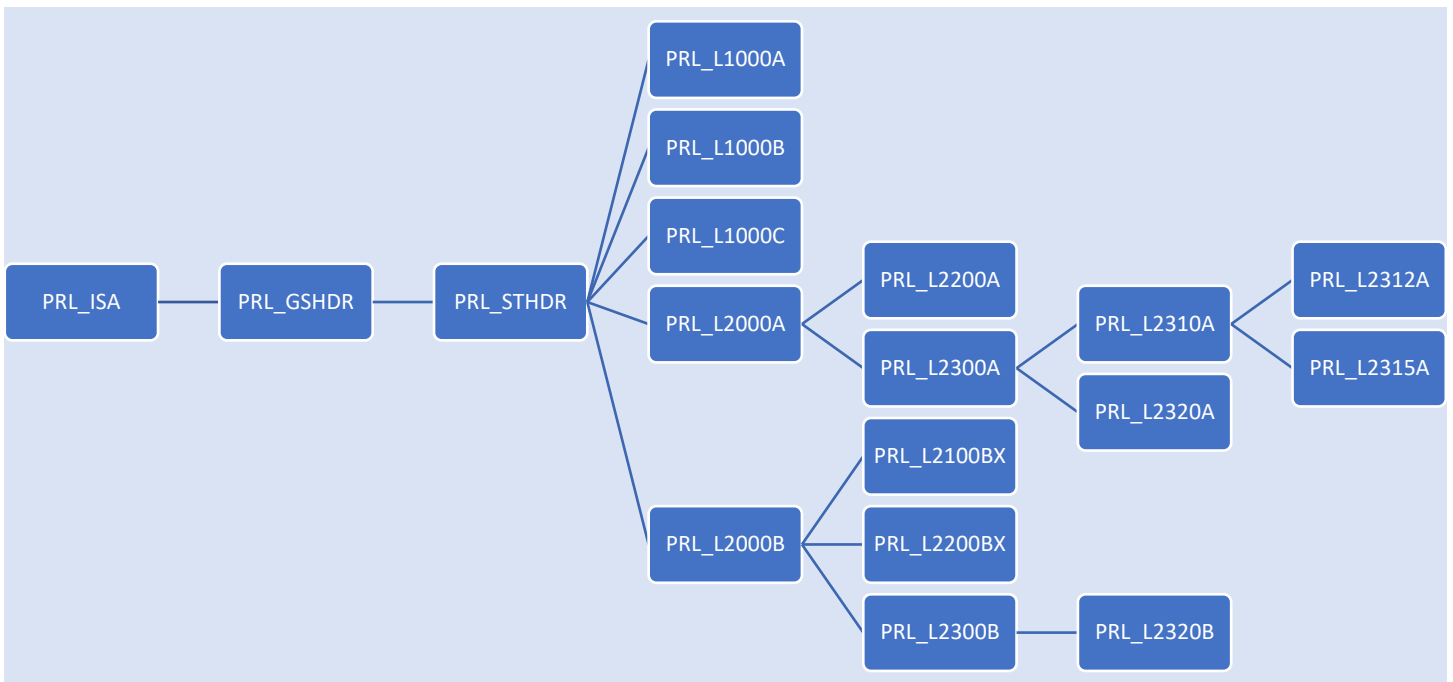


5010_278_ACK / R1 Health Care Services Review - Acknowledgment / SVK

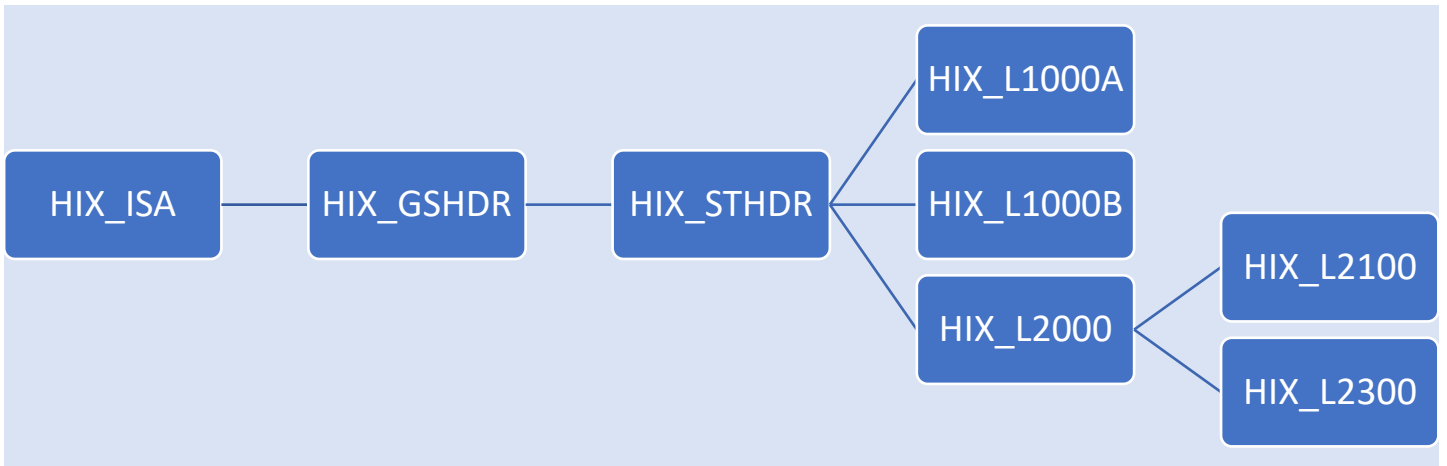


5010_820 / S0

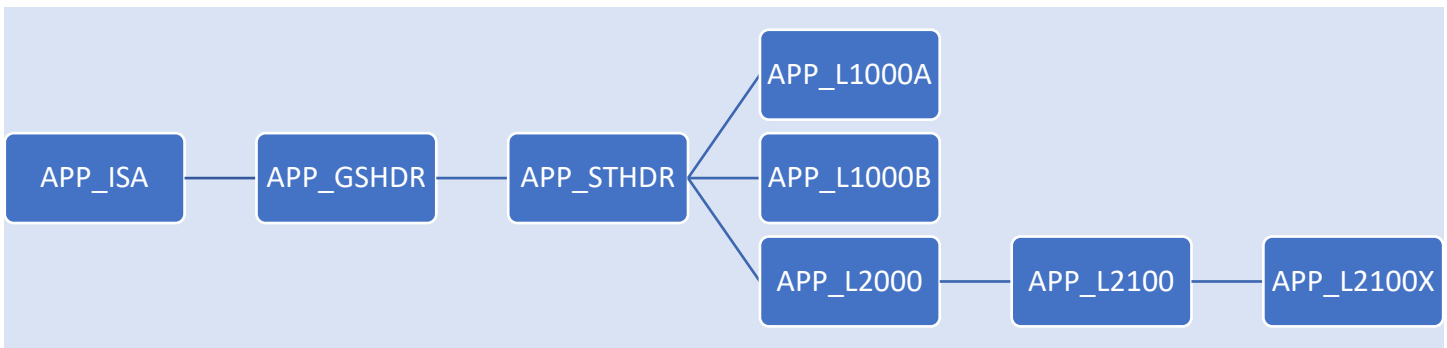
Payroll Deducted and Other Group Premium Payment for Insurance Products / PRL

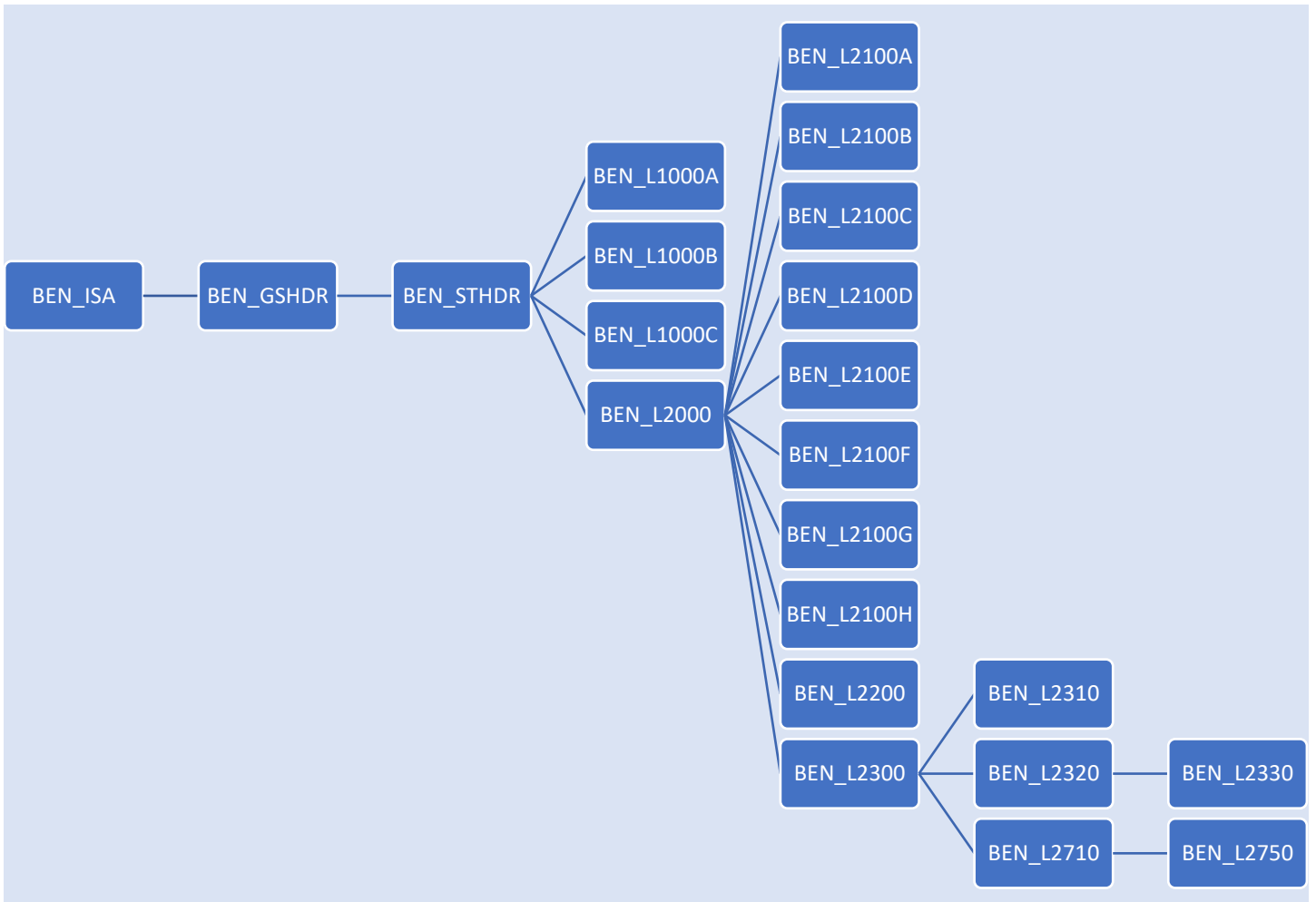


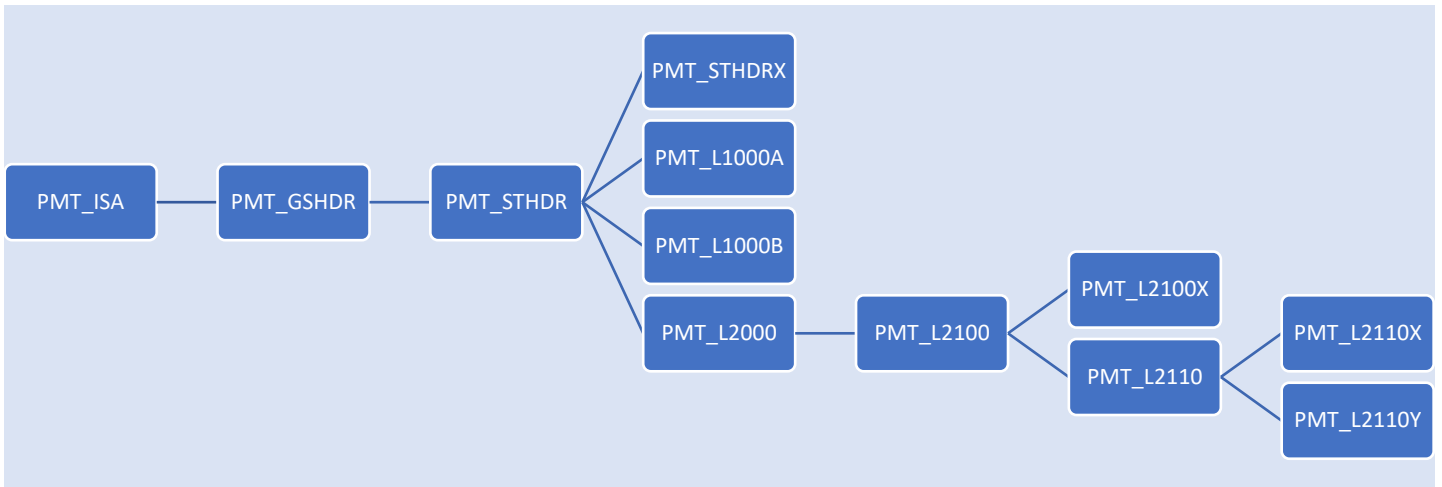
5010_820X / S5 Health Insurance Exchange Related Payments / HIX

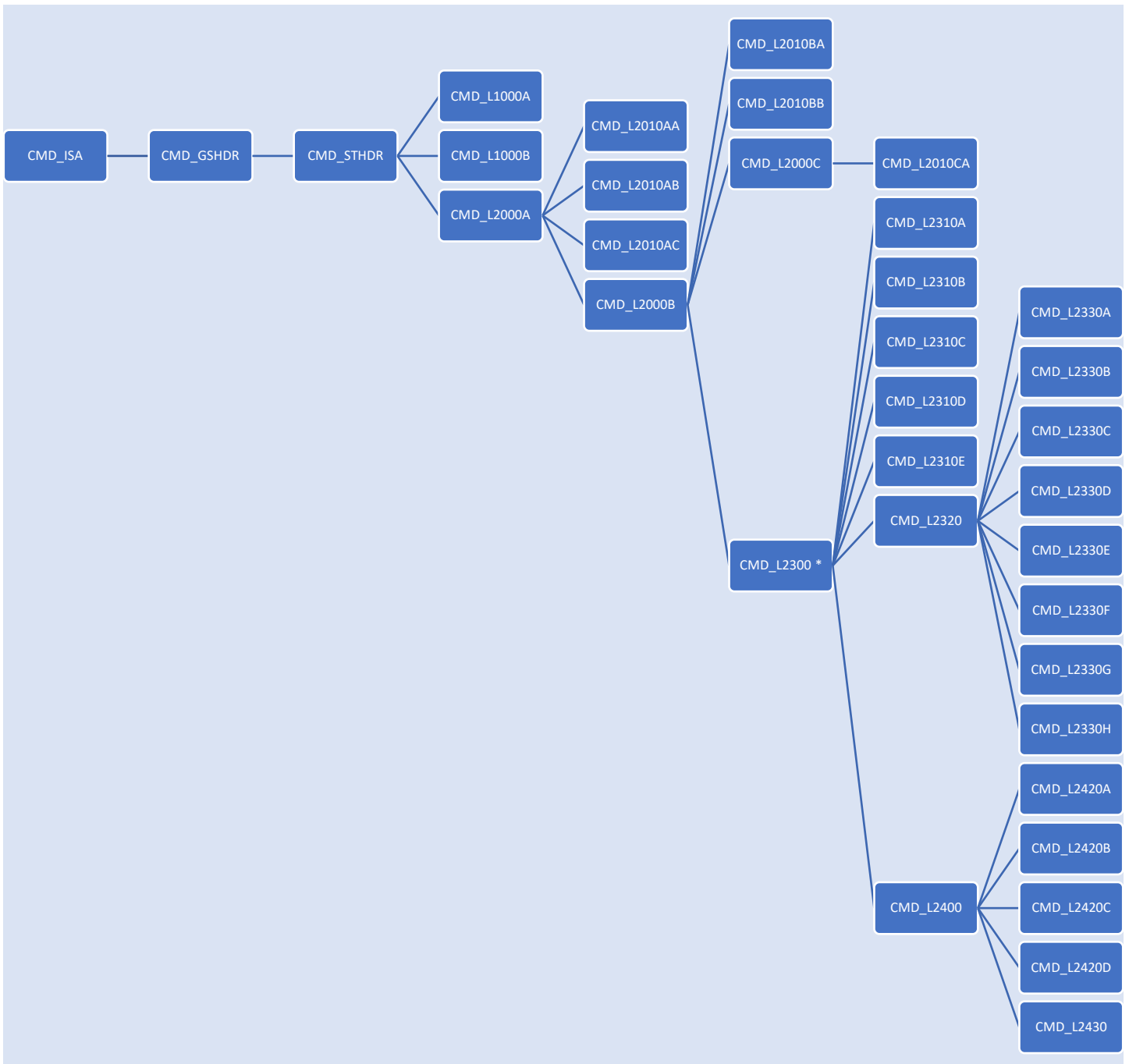


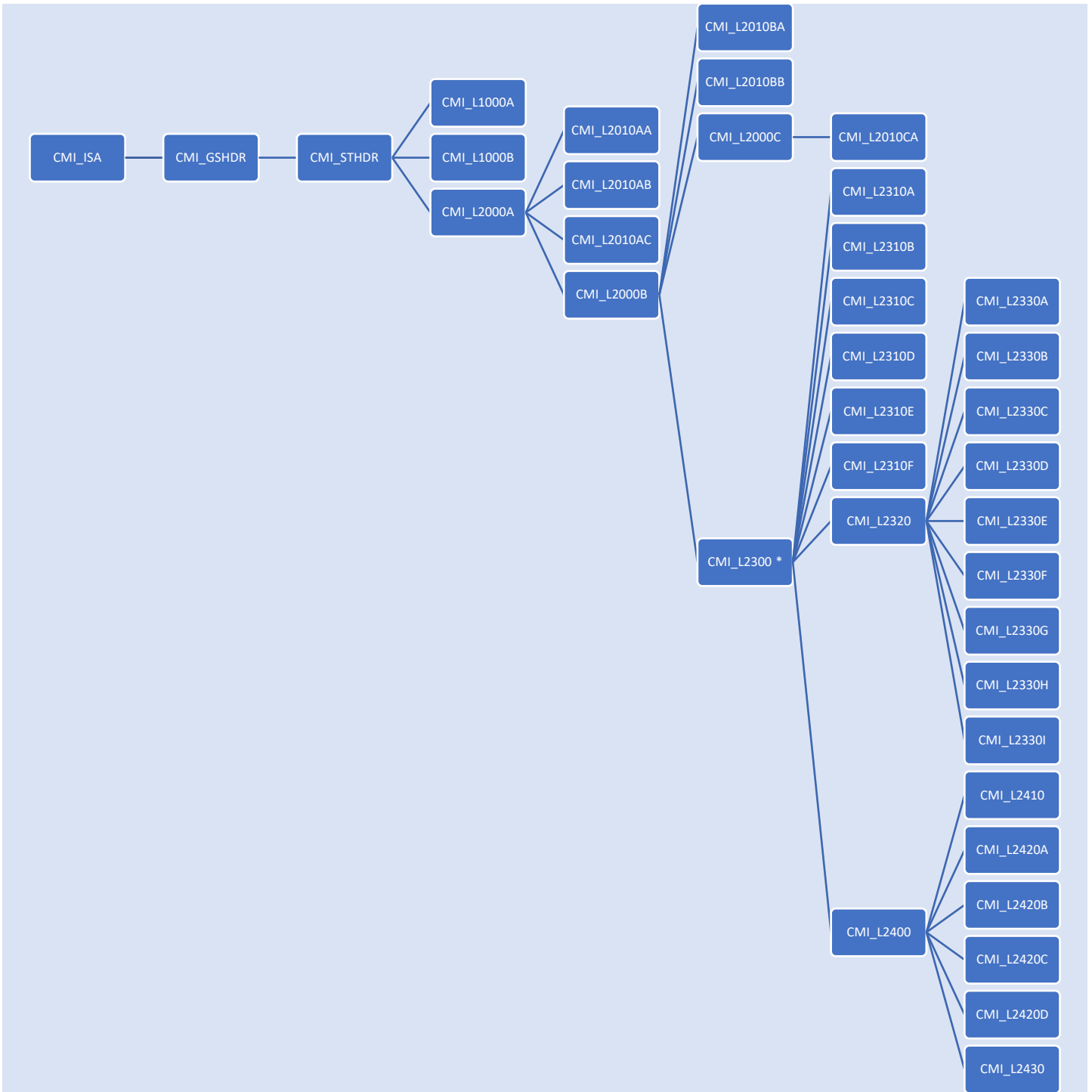
5010_824 / P7 Application Reporting for Insurance / APP

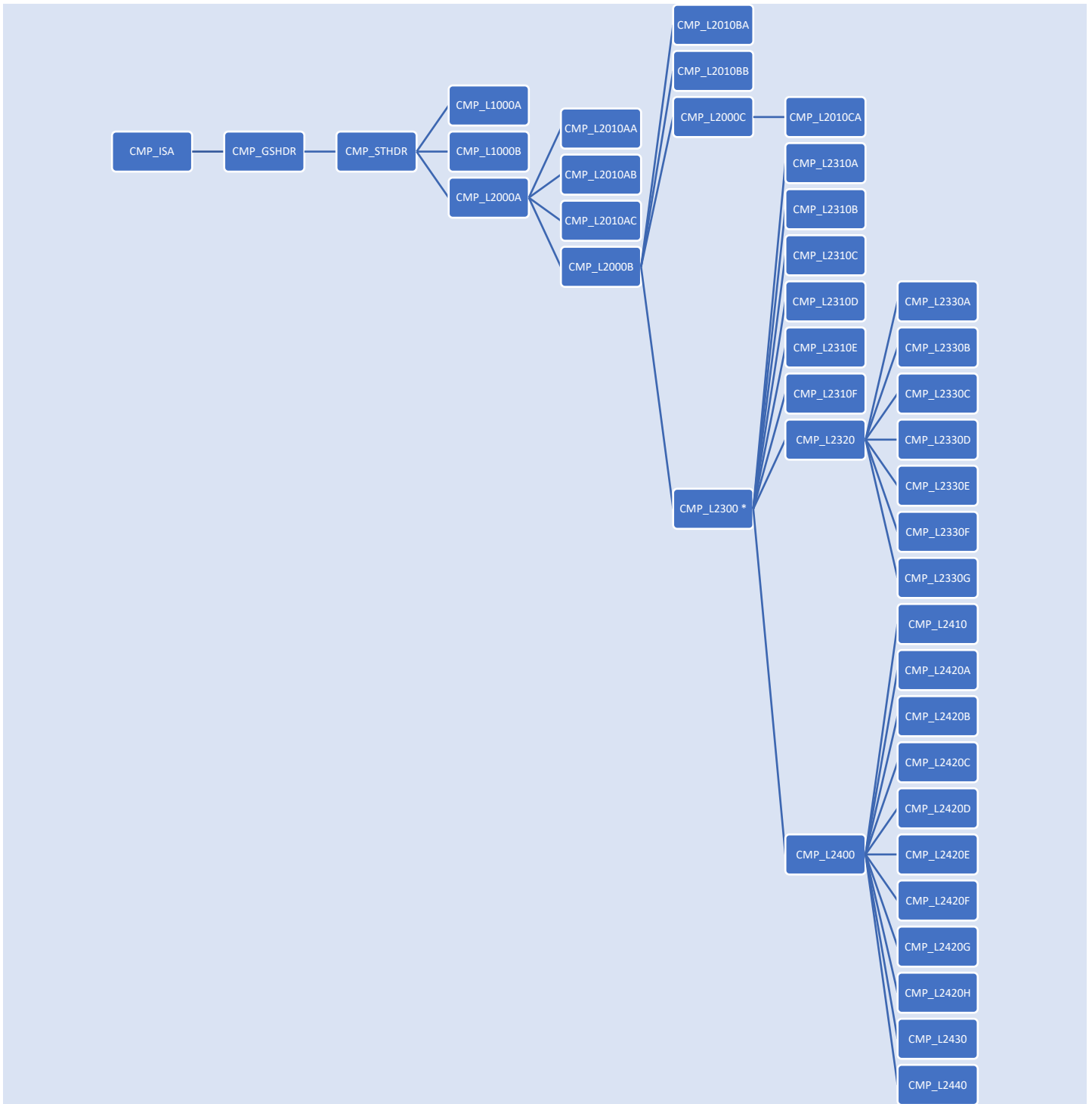












* The L2300 tables in the 837 D/I/P specifications have *two* parent indexes – one for 2000B as shown in these diagrams, and an *additional, optional* PAR_2000C_IX field that relates the claim to a specific iteration of the patient loop. If there are no L2000C patient loops, then this field will be null.

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

Creating Outbound Transactions

The SEED system within SERENEDI is a set of stored procedures and fixed data that enable you to create new, non-PHI test files for 13 of the 18 supported 5010 EDI specifications. These files provide a fixed starting point from which to begin development, using well-known SQL syntax and objects.

Because 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. The 13 stored procedures 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 Flat 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.REFERENCE.DOCS/specs folder.

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 segments or highly dynamic mappings – may be much easier to express using the XML format than the Flat database format.

The Functionality Walkthrough provides examples for kicking off the conversion to create EDI files – here is an example for creating a 270 file from the extract:

```
-- 270 - Eligibility Inquiry
INSERT INTO SERENEDI.MAIN.BIZ_EVENT
(BIZ_EVENT_ID, BIZ_TRIGGER_ID, EVENT_DATE,
EVENT_CRIT, SOURCE_NM, EVENT_DATA1,
```

```

EVENT_DATA2)
SELECT SERENEDI.MAIN.BIZ_EVENT_SEQ.NEXTVAL,
8, CURRENT_TIMESTAMP(), NULL, 'USER',
'CALL SERENEDI.MAIN.USP_270_EXTRACT()',
'@SERENEDI_DATA.STAGES.EDI_OUT/EDI_FROM_BIN/TEST_270.txt';

CALL SERENEDI.MAIN.HEARTBEAT();

```

Once this sproc is called, the EDI file will be waiting in the location specified in the EVENT_DATA2 argument.

Tips for Creating Outbound EDI Files

1. 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 that you expect.
2. 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. Encoding the wrong data type into a map can have unpredictable effects.
3. 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.
4. 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 13 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 along with the source code of that extract.

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.

-- SPROC: USP_270_EXTRACT

CREATE OR REPLACE PROCEDURE MAIN.USP_270_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN

-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_270';

-- Extraction

LET result RESULTSET := (
SELECT ISA_ISA02_NO_AUTH_NFO AS M1_ISA_ISA02_NO_AUTH_NFO,
ISA_ISA04_PSSWD,
ISA_ISA06_MUTLY_DEF,
ISA_ISA08_MUTLY_DEF,
ISA_ISA11_REPTN_SEP,
ISA_ISA12_ICN_VERS_NR,
ISA_ISA13_ICN,
ISA_ISA15_ICN_USG_IND,
ISA_ISA16_COMP_ELE_SEP,
ISA_ISA13_ICN AS ISA_IEA02_ICN,
GSHDR_GS02_APP_SNDR_CD,
GSHDR_GS03_APP_RCV_CD,
GSHDR_GS06_GCN,
GSHDR_GS06_GCN AS GSHDR_GE02_GCN,

''10000000'' AS STHDR_ST02_TCN,
''0022'' AS STHDR_BHT01_HIER_STRUC_CD,
''13'' AS STHDR_BHT02_TS_PURP_CD,
''10000000'' AS STHDR_BHT03_SBM_TRANS_ID,
CURRENT_TIMESTAMP() AS STHDR_BHT04_TS_CRTN_D8,
CURRENT_TIMESTAMP() AS STHDR_BHT05_TS_CRTN_TM,
''10000000'' AS STHDR_SE02_TCN,
P1.PROV_ORG_NM AS L2100A_PR_NM103_NONPSNENT_NM,
P1.PROV_NPI AS L2100A_PR_NM109_NPI,
P2.PROV_ORG_NM AS L2100B_1P_NM103_NONPSNENT_NM,
P2.PROV_NPI AS L2100B_1P_NM109_NPI,
P2.TAX_ID AS L2100B_1P_REF_TAX_ID,
P2.BIZ_ADDR AS L2100B_1P_N301_NFO_RCVR_ADDR,
P2.BIZ_CITY AS L2100B_1P_N401_NFO_RCV_CITY_NM,
P2.BIZ_STATE AS L2100B_1P_N402_NFO_RCV_STAT,
P2.BIZ_ZIP AS L2100B_1P_N403_NFO_RCV_ZIP,
M1.LAST_NM AS L2100C_NM103_PERSN,
M1.FIRST_NM AS L2100C_NM104_SBR_FNM,
M1.MEM_ID AS L2100C_NM109_MEM_ID_NR,
M1.SSN AS L2100C_REF_SSN,
M1.RES_ADDR AS L2100C_N301_SBR_ADDR,
M1.RES_CITY AS L2100C_N401_SBR_CITY,
M1.RES_STATE AS L2100C_N402_SBR_STAT,
M1.RES_ZIP AS L2100C_N403_SBR_ZIP,
M1.DOB AS L2100C_DMG02_D8,
M1.GENDER AS L2100C_DMG03_SUB_GENDR_CD,
''30'' AS L2110C_EQ01_E01_SVC_TYP_CD,

M2.LAST_NM AS L2100D_NM103_DEP_LNM,
M2.FIRST_NM AS L2100D_NM104_DEP_FNM,
M2.SSN AS L2100D_REF_SSN,
M2.DOB AS L2100D_DMG02_D8,

```

        M2.GENDER                AS L2100D_DMG03_DEP_GNDR_CD,
        '30'                      AS L2110D_EQ01_E01_SVC_TYP_CD,
        1                          AS NEWROW
FROM   SAMPL_HEADER SH
INNER JOIN (
        SELECT *, ROW_NUMBER() OVER (ORDER BY MEMBER_ID) AS RN
        FROM SAMPL_MEMBER
        WHERE RELATION = 'P'
    ) M1
    ON M1.RN <= 40
INNER JOIN SAMPL_PROVIDER P1
    ON P1.PROVIDER_ID = M1.PAYER_PROVIDER_ID
INNER JOIN SAMPL_PROVIDER P2
    ON P2.PROVIDER_ID = M1.BILLER_PROVIDER_ID
INNER JOIN SAMPL_MEMBER M2
    ON M2.PAR_MEMBER_ID = M1.MEMBER_ID
    AND M2.RELATION = 'D'
WHERE SH.HDR_NAME = 'XPT_270'
ORDER BY L2100A_PR_NM109_NPI,
         L2100B_1P_NM109_NPI,
         L2100C_NM109_MEM_ID_NR
);

RETURN TABLE(result);
END;
';

```

USP_271_EXTRACT

The 271 extract is almost exactly the same as the 270 extract, except that it adds a Plan Begin date of 2019-01-01 for all of the 120 records it returns.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_271_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_271';

-- Extraction
LET result RESULTSET := (
    SELECT  ISA_ISA02_NO_AUTH_NFO      AS N1_ISA_ISA02_NO_AUTH_NFO,
            ISA_ISA04_PSSWD,
            ISA_ISA06_MUTLY_DEF,
            ISA_ISA08_MUTLY_DEF,
            ISA_ISA11_REPTN_SEP,
            ISA_ISA12_ICN_VERS_NR,
            ISA_ISA13_ICN,
            ISA_ISA15_ICN_USG_IND,
            ISA_ISA16_COMP_ELE_SEP,
            ISA_ISA13_ICN              AS ISA_IEA02_ICN,
            GSHDR_GS02_APP_SNDR_CD,
            GSHDR_GS03_APP_RCV_CD,
            GSHDR_GS06_GCN,
            GSHDR_GS06_GCN            AS GSHDR_GE02_GCN,
            '10000000'                AS STHDR_ST02_TCN,
            '0022'                    AS STHDR_BHT01_HIER_STRUC_CD,
            '11'                      AS STHDR_BHT02_TS_PURP_CD,
            '10000000'                AS STHDR_BHT03_SBM_TRANS_ID,

```

```

CURRENT_TIMESTAMP()      AS STHDR_BHT04_TS_CRTN_D8,
CURRENT_TIMESTAMP()      AS STHDR_BHT05_TS_CRTN_TM,
''100000000''           AS STHDR_SE02_TCN,
P1.PROV_ORG_NM           AS L2100A_PR_NM103_NONPSNENT_NM,
P1.PROV_NPI              AS L2100A_PR_NM109_NPI,
                          AS L2100B_1P_NM102_ENT_TYP_QUAL,
P2.PROV_ORG_NM           AS L2100B_1P_NM103_NFO_RCV_NM,
P2.PROV_NPI              AS L2100B_1P_NM109_NPI,
P2.TAX_ID                AS L2100B_1P_REF_TAX_ID,
M1.LAST_NM               AS L2100C_NM103_SBR_LNM,
M1.FIRST_NM              AS L2100C_NM104_SBR_FNM,
M1.MEM_ID                AS L2100C_NM109_MEM_ID_NR,
M1.SSN                   AS L2100C_REF_SSN,
M1.RES_ADDR              AS L2100C_N301_SBR_ADDR,
M1.RES_CITY              AS L2100C_N401_SBR_CITY,
M1.RES_STATE             AS L2100C_N402_SBR_STAT,
M1.RES_ZIP               AS L2100C_N403_SBR_ZIP,
M1.DOB                   AS L2100C_DMG02_D8,
M1.GENDER                AS L2100C_DMG03_SUB_GENDR_CD,
''1''                   AS L2110C_EB01_E01_ELG_BEN_NFO,
''30''                   AS L2110C_EB03_E01_SVC_TYP_CD,
TO_TIMESTAMP_NTZ(''2019-01-01'')
                          AS L2110C_DTP_PLAN_BGN_D8,

M2.LAST_NM               AS L2100D_NM103_DEP_LNM,
M2.FIRST_NM              AS L2100D_NM104_DEP_FNM,
M2.SSN                   AS L2100D_REFB_SSN,
M2.DOB                   AS L2100D_DMG02_D8,
M2.GENDER                AS L2100D_DMG03_DEP_GNDR_CD,
''1''                   AS L2110D_EB01_E01_ELG_BEN_NFO,
''30''                   AS L2110D_EB03_E01_SVC_TYP_CD,
TO_TIMESTAMP_NTZ(''2019-01-01'')
                          AS L2110D_DTP_PLAN_BGN_D8,
1                          AS NEWROW
FROM   SAMPL_HEADER SH
INNER JOIN SAMPL_MEMBER M1
ON     M1.RELATION = 'P'
INNER JOIN SAMPL_PROVIDER P1
ON     P1.PROVIDER_ID = M1.PAYER_PROVIDER_ID
INNER JOIN SAMPL_PROVIDER P2
ON     P2.PROVIDER_ID = M1.BILLER_PROVIDER_ID
INNER JOIN SAMPL_MEMBER M2
ON     M2.PAR_MEMBER_ID = M1.MEMBER_ID
AND    M2.RELATION = 'D'
WHERE  SH.HDR_NAME = 'XPT_271'
ORDER BY L2100A_PR_NM109_NPI,
         L2100B_1P_NM109_NPI,
         L2100C_NM109_MEM_ID_NR
);

RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_276_EXTRACT()
RETURNS TABLE ( )
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_276';

-- Extraction
LET result RESULTSET := (
    SELECT  ISA_ISA02_NO_AUTH_NFO      AS 00_ISA_ISA02_NO_AUTH_NFO,
            ISA_ISA04_PSSWD,
            ISA_ISA06_MUTLY_DEF,
            ISA_ISA08_MUTLY_DEF,
            ISA_ISA11_REPTN_SEP,
            ISA_ISA12_ICN_VERS_NR,
            ISA_ISA13_ICN,
            ISA_ISA15_ICN_USG_IND,
            ISA_ISA16_COMP_ELE_SEP,
            ISA_ISA13_ICN              AS ISA_IEA02_ICN,

            GSHDR_GS02_APP_SNDR_CD,
            GSHDR_GS03_APP_RCV_CD,
            GSHDR_GS06_GCN,
            GSHDR_GS06_GCN            AS GSHDR_GE02_GCN,

            '10000000'                AS STHDR_ST02_TCN,
            '10000000'                AS STHDR_BHT03_REF_ID,
            LOCALTIMESTAMP()          AS STHDR_BHT04_TS_CRTN_D8,
            TO_TIMESTAMP_NTZ('1800-01-01 ' || TO_CHAR(LOCALTIMESTAMP(), 'HH24:MI:SS'))
            AS STHDR_BHT05_TS_CRTN_TM,
            '10000000'                AS STHDR_SE02_TCN,

            P1.PROV_ORG_NM             AS L2100A_NM103_NONPSNENT_NM,
            P1.PROV_NPI                AS L2100A_NM109_PAYR_ID,

            '2'                        AS L2100B_NM102_ENT_TYP_QUAL,
            P2.PROV_NPI                AS L2100B_NM103_NFO_RCV_NM,
            P2.TAX_ID                  AS L2100B_NM109_ETN_NR,

            '2'                        AS L2100C_01_NM102_ENT_TYP_QUAL,
            P2.PROV_ORG_NM             AS L2100C_01_NM103_PVR_LNM,
            P2.PROV_NPI                AS L2100C_01_NM109_NPI,

            DOB                       AS L2000D_DMG02_D8,
            GENDER                     AS L2000D_DMG03_SUB_GENDR_CD,
            LAST_NM                     AS L2100D_NM103_PERSN_LNM,
            FIRST_NM                    AS L2100D_NM104_SBR_FNM,
            IDCD                       AS L2100D_NM109_MEM_ID_NR,
            CASE WHEN DOB1 IS NULL THEN C.CLM_NR ELSE NULL END
            AS L2200D_TRN02_CUR_TRANS_TRAC_NR,
            CASE WHEN DOB1 IS NULL THEN C.CLM_NR ELSE NULL END

```

```

CASE WHEN DOB1 IS NULL THEN AS L2200D_REF_PYR_CLM_NR,
D.HCPCS_CD ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_SVC0102_HCPCS_CD,
D.MOD01 ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_SVC0103_PROC_MOD,
D.MOD02 ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_SVC0104_PROC_MOD,
D.CHG_AMT::FLOAT ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_SVC02_LIN_ITM_CHG_AMT,
D.UNITS::FLOAT ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_SVC07_UN_SVC_CT,
D.LINE_SEQ::VARCHAR ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_REF_LIN_ITM,
D.SVC_DT ELSE NULL END
CASE WHEN DOB1 IS NULL THEN AS L2210D_DTP_SVC_D8,

DOB1 AS L2000E_DMG02_D8,
GENDER1 AS L2000E_DMG03_PAT_GNDR_CD,
LAST_NM1 AS L2100E_NM103_PERSN_LNM,
FIRST_NM1 AS L2100E_NM104_PT_FNM,
CASE WHEN DOB1 IS NOT NULL THEN C.CLM_NR ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2200E_TRN02_CUR_TRANS_TRAC_NR,
C.CLM_NR ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2200E_REF_PYR_CLM_NR,
D.HCPCS_CD ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2210E_SVC0102_HCPCS_CD,
D.MOD01 ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2210E_SVC0103_PROC_MOD,
D.MOD02 ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2210E_SVC0104_PROC_MOD,
D.CHG_AMT::FLOAT ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2210E_SVC02_LIN_ITM_CHG_AMT,
D.UNITS::FLOAT ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2210E_SVC07_UN_SVC_CT,
D.SVC_DT ELSE NULL END
CASE WHEN DOB1 IS NOT NULL THEN AS L2210E_DTP_SVC_D8,
1 AS NEWROW

```

FROM

```

( SELECT MEMBER_ID AS PAR_MEM_ID,
NULL AS MEMBER_ID,
PAYER_PROVIDER_ID,
BILLER_PROVIDER_ID,
DOB,
GENDER,
LAST_NM,
FIRST_NM,
MEM_ID AS IDCD,
NULL AS DOB1,
NULL AS GENDER1,
NULL AS LAST_NM1,
NULL AS FIRST_NM1,
NULL AS IDCD1

```

```

FROM SAMPL_MEMBER
WHERE RELATION = 'P'

```

UNION

```

SELECT M2.MEMBER_ID AS PAR_MEM_ID,
M1.MEMBER_ID AS MEMBER_ID,
M1.PAYER_PROVIDER_ID,
M1.BILLER_PROVIDER_ID,
NULL,
NULL,
M2.LAST_NM,

```

```

                M2.FIRST_NM,
                M2.MEM_ID      AS IDCD,
                M1.DOB         AS DOB1,
                M1.GENDER      AS GENDER1,
                M1.LAST_NM     AS LAST_NM1,
                M1.FIRST_NM    AS FIRST_NM1,
                M1.MEM_ID      AS IDCD1
            FROM    SAMPL_MEMBER M1
            INNER JOIN SAMPL_MEMBER M2
            ON      M2.PAR_MEMBER_ID = M1.MEMBER_ID
            WHERE   M1.RELATION = 'P'
            AND     M2.RELATION = 'D'
        ) MEM

        INNER JOIN
            SAMPL_PROVIDER P1
        ON    P1.PROVIDER_ID = MEM.PAYER_PROVIDER_ID

        INNER JOIN
            SAMPL_PROVIDER P2
        ON    P2.PROVIDER_ID = MEM.BILLER_PROVIDER_ID

        INNER JOIN
            SAMPL_CLAIM C
        ON    C.MEMBER_ID = MEM.PAR_MEM_ID

        INNER JOIN
            SAMPL_CLAIM_DTL D
        ON    D.CLAIM_ID = C.CLAIM_ID

        INNER JOIN
            SAMPL_HEADER SH
        ON    SH.HDR_NAME = 'XPT_276'

        ORDER BY
            MEM.PAR_MEM_ID
    );

    RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_277_EXTRACT()
RETURNS TABLE()
LANGUAGE SQL
EXECUTE AS OWNER
AS
$$
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_277';

-- Extraction
LET result RESULTSET := (
    SELECT  ISA_ISA02_NO_AUTH_NFO      AS P0_ISA_ISA02_NO_AUTH_NFO,

```

```

ISA_ISA04_PSSWD,
ISA_ISA06_MUTLY_DEF,
ISA_ISA08_MUTLY_DEF,
ISA_ISA11_REPTN_SEP,
ISA_ISA12_ICN_VERS_NR,
ISA_ISA13_ICN,
ISA_ISA15_ICN_USG_IND,
ISA_ISA16_COMP_ELE_SEP,
ISA_ISA13_ICN          AS ISA_IEA02_ICN,

GSHDR_GS02_APP_SNDR_CD,
GSHDR_GS03_APP_RCV_CD,
GSHDR_GS06_GCN,
GSHDR_GS06_GCN          AS GSHDR_GE02_GCN,

'100000000'          AS STHDR_ST02_TCN,
'100000000'          AS STHDR_BHT03_ORIG_APP_TRANS_ID,
LOCALTIMESTAMP()    AS STHDR_BHT04_TS_CRTN_D8,
TO_TIMESTAMP_NTZ('1800-01-01 ' || TO_CHAR(LOCALTIMESTAMP(), 'HH24:MI:SS'))
AS STHDR_BHT05_TS_CRTN_TM,
'100000000'          AS STHDR_SE02_TCN,

P1.PROV_ORG_NM        AS L2100A_NM103_NONPSNENT_NM,
P1.PROV_NPI           AS L2100A_NM109_PAYR_ID,

'2'                   AS L2100B_NM102_ENT_TYP_QUAL,
P2.PROV_ORG_NM        AS L2100B_NM103_NFO_RCV_NM,
P2.TAX_ID             AS L2100B_NM109_ETN_NR,

'2'                   AS L2100C_01_NM102_ENT_TYP_QUAL,
P2.PROV_ORG_NM        AS L2100C_01_NM103_PVR_LNM,
P2.PROV_NPI           AS L2100C_01_NM109_NPI,

MEM.LAST_NM           AS L2100D_NM103_PERSN_LNM,
MEM.FIRST_NM          AS L2100D_NM104_SBR_FNM,
MEM.IDCD              AS L2100D_NM109_MEM_ID_NR,

CASE WHEN IDCD1 IS NULL THEN C.CLM_NR ELSE NULL END
AS L2200D_TRN02_REF_TRAN_TRAC_NR,
CASE WHEN IDCD1 IS NULL THEN 'A1' ELSE NULL END
AS L2200DX_STC0101_HTCRCLM_CAT_CD,
CASE WHEN IDCD1 IS NULL THEN '20' ELSE NULL END
AS L2200DX_STC0102_STATS_CD,
CASE WHEN IDCD1 IS NULL THEN CURRENT_TIMESTAMP() ELSE NULL END
AS L2200DX_STC02_STMT_NFO_EFF_D8,
CASE WHEN IDCD1 IS NULL THEN C.CLM_NR ELSE NULL END
AS L2200D_REF_PYR_CLM_NR,
CASE WHEN IDCD1 IS NULL THEN D.HCPCS_CD ELSE NULL END
AS L2220D_SVC0102_HCPCS_CD,
CASE WHEN IDCD1 IS NULL THEN D.MOD01 ELSE NULL END
AS L2220D_SVC0103_PROC_MOD,
CASE WHEN IDCD1 IS NULL THEN D.MOD02 ELSE NULL END
AS L2220D_SVC0104_PROC_MOD,
CASE WHEN IDCD1 IS NULL THEN D.CHG_AMT::FLOAT ELSE NULL END
AS L2220D_SVC02_LIN_ITM_CHG_AMT,
CASE WHEN IDCD1 IS NULL THEN D.PMT_AMT::FLOAT ELSE NULL END
AS L2220D_SVC03_LIN_ITM_PMT_AMT,
CASE WHEN IDCD1 IS NULL THEN D.UNITS::FLOAT ELSE NULL END
AS L2220D_SVC07_UN_SVC_CT,
CASE WHEN IDCD1 IS NULL THEN 'A1' ELSE NULL END
AS L2220DX_STC0101_HTCRCLM_CAT_CD,
CASE WHEN IDCD1 IS NULL THEN '20' ELSE NULL END
AS L2220DX_STC0102_STATS_CD,
CASE WHEN IDCD1 IS NULL THEN CURRENT_TIMESTAMP() ELSE NULL END
AS L2220DX_STC02_STMT_NFO_EFF_D8,
CASE WHEN IDCD1 IS NULL THEN D.SVC_DT ELSE NULL END

```

```

AS L2220D_DTP_SVC_D8,
CASE WHEN IDCD1 IS NOT NULL THEN MEM.LAST_NM1 ELSE NULL END
AS L2100E_NM103_PT_LNM,
CASE WHEN IDCD1 IS NOT NULL THEN MEM.FIRST_NM1 ELSE NULL END
AS L2100E_NM104_PT_FNM,
CASE WHEN IDCD1 IS NOT NULL THEN 'XXX' ELSE NULL END
AS L2200E_TRN02_REF_TRAN_TRAC_NR,
CASE WHEN IDCD1 IS NOT NULL THEN 'A1' ELSE NULL END
AS L2200EX_STC0101_HTCRCLM_CAT_CD,
CASE WHEN IDCD1 IS NOT NULL THEN '20' ELSE NULL END
AS L2200EX_STC0102_STATS_CD,
CASE WHEN IDCD1 IS NOT NULL THEN CURRENT_TIMESTAMP() ELSE NULL END
AS L2200EX_STC02_STMT_NFO_EFF_D8,
CASE WHEN IDCD1 IS NOT NULL THEN D.HCPCS_CD ELSE NULL END
AS L2220E_SVC0102_HCPCS_CD,
CASE WHEN IDCD1 IS NOT NULL THEN D.MOD01 ELSE NULL END
AS L2220E_SVC0103_PROC_MOD,
CASE WHEN IDCD1 IS NOT NULL THEN D.MOD02 ELSE NULL END
AS L2220E_SVC0104_PROC_MOD,
CASE WHEN IDCD1 IS NOT NULL THEN D.CHG_AMT::FLOAT ELSE NULL END
AS L2220E_SVC02_LIN_ITM_CHG_AMT,
CASE WHEN IDCD1 IS NOT NULL THEN D.PMT_AMT::FLOAT ELSE NULL END
AS L2220E_SVC03_LIN_ITM_PMT_AMT,
CASE WHEN IDCD1 IS NOT NULL THEN D.UNITS::FLOAT ELSE NULL END
AS L2220E_SVC07_UN_SVC_CT,

CASE WHEN IDCD1 IS NOT NULL THEN 'A1' ELSE NULL END
AS L2220EX_STC0101_HTCRCLM_CAT_CD,
CASE WHEN IDCD1 IS NOT NULL THEN '20' ELSE NULL END
AS L2220EX_STC0102_STATS_CD,
CASE WHEN IDCD1 IS NOT NULL THEN CURRENT_TIMESTAMP() ELSE NULL END
AS L2220EX_STC02_STMT_NFO_EFF_D8,

CASE WHEN IDCD1 IS NOT NULL THEN C.CLM_NR ELSE NULL END
AS L2220E_REF_LIN_ITM,
CASE WHEN IDCD1 IS NOT NULL THEN D.SVC_DT ELSE NULL END
AS L2220E_DTP_SVC_D8,

```

```

1 AS NEWROW

```

FROM

```

(
SELECT MEMBER_ID AS PAR_MEM_ID,
NULL AS MEMBER_ID,
PAYER_PROVIDER_ID,
BILLER_PROVIDER_ID,
LAST_NM,
FIRST_NM,
MEM_ID AS IDCD,
NULL AS LAST_NM1,
NULL AS FIRST_NM1,
NULL AS IDCD1

```

```

FROM SAMPL_MEMBER
WHERE RELATION = 'P'

```

UNION ALL

```

SELECT M2.MEMBER_ID AS PAR_MEM_ID,
M1.MEMBER_ID AS MEMBER_ID,
M1.PAYER_PROVIDER_ID,
M1.BILLER_PROVIDER_ID,
M1.LAST_NM,
M1.FIRST_NM,
M1.MEM_ID AS IDCD,
M2.LAST_NM AS LAST_NM1,
M2.FIRST_NM AS FIRST_NM1,
M2.MEM_ID AS IDCD1

```

```

        FROM    SAMPL_MEMBER M1
        INNER JOIN SAMPL_MEMBER M2
              ON M2.PAR_MEMBER_ID = M1.MEMBER_ID
        WHERE   M1.RELATION = 'P'
        AND     M2.RELATION = 'D'
    ) MEM

    INNER JOIN SAMPL_PROVIDER P1
        ON P1.PROVIDER_ID = MEM.BILLER_PROVIDER_ID

    INNER JOIN SAMPL_PROVIDER P2
        ON P2.PROVIDER_ID = MEM.PAYER_PROVIDER_ID

    INNER JOIN SAMPL_CLAIM C
        ON C.MEMBER_ID = MEM.PAR_MEM_ID

    INNER JOIN SAMPL_CLAIM_DTL D
        ON D.CLAIM_ID = C.CLAIM_ID

    INNER JOIN SAMPL_HEADER SH
        ON SH.HDR_NAME = 'XPT_277'

    ORDER BY
        MEM.PAR_MEM_ID DESC,
        MEM.MEMBER_ID
);

RETURN TABLE(result);
END;
$$;

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_277CA_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_277CA';

-- Extraction
LET result RESULTSET := (
    SELECT  ISA_ISA02_NO_AUTH_NFO          AS P5_ISA_ISA02_NO_AUTH_NFO,
            ISA_ISA04_PSSWD,
            ISA_ISA06_MUTLY_DEF,
            ISA_ISA08_MUTLY_DEF,
            ISA_ISA11_REPTN_SEP,
            ISA_ISA12_ICN_VERS_NR,
            ISA_ISA13_ICN,
            ISA_ISA15_ICN_USG_IND,
            ISA_ISA16_COMP_ELE_SEP,
            ISA_ISA13_ICN                  AS ISA_IEA02_ICN,

```

GSHDR_GS02_APP_SNDR_CD,
 GSHDR_GS03_APP_RCV_CD,
 GSHDR_GS06_GCN,
 GSHDR_GS06_GCN AS GSHDR_GE02_GCN,

(10000000 + BLLR.BILLER_PROVIDER_ID)::VARCHAR AS STHDR_ST02_TCN,
 (10000000 + BLLR.BILLER_PROVIDER_ID)::VARCHAR AS STHDR_BHT03_REF_ID,
 CURRENT_TIMESTAMP() AS STHDR_BHT04_TS_CRTN_D8,
 CURRENT_TIMESTAMP() AS STHDR_BHT05_TS_CRTN_TM,
 (10000000 + BLLR.BILLER_PROVIDER_ID)::VARCHAR AS STHDR_SE02_TCN,

P1.PROV_ORG_NM AS L2100A_PR_NM103_NONPSNENT_NM,
 P1.PROV_NPI AS L2100A_PR_NM109_PAYR_ID,

''10000000'' AS L2200A_PR_TRN02_NFO_SRC_AP_TRC,
 DATEADD(DAY, -1, LOCALTIMESTAMP()) AS L2200A_PR_DTP_RCVD_D8,
 LOCALTIMESTAMP() AS L2200A_PR_DTP_PRCS_D8,

P2.PROV_ORG_NM AS L2100B_NM103_NONPSNENT_NM,
 P2.TAX_ID AS L2100B_NM109_ETN_NR,

''1'' AS L2200B_TRN02_CLM_TRNS_BAT_NR,

''A1'' AS L2200BX_STC0101_HTCRCLM_CAT_CD,
 ''20'' AS L2200BX_STC0102_HTCRCLM_STATCD,
 LOCALTIMESTAMP() AS L2200BX_STC02_STMT_NFO_EFF_D8,
 ''WQ'' AS L2200BX_STC03_ACTN_CD,
 (SELECT SUM(CHG_AMT)::FLOAT FROM SAMPL_CLAIM_DTL) AS L2200BX_STC04_TOT_SBR_CHG_WK,

(SELECT COUNT(*)::FLOAT FROM SAMPL_CLAIM) AS L2200B_QTY02_ACK_QTY,
 (SELECT SUM(CHG_AMT)::FLOAT FROM SAMPL_CLAIM_DTL) AS L2200B_AMT02_IN_PROCS,

P2.PROV_ORG_NM AS L2100C_NM103_NONPSNENT_NM,
 P2.PROV_NPI AS L2100C_NM109_NPI,

''0'' AS L2200C_TRN02_PVR_SVCNFO_TRC_ID,
 ''A1'' AS L2200CX_STC0101_HTCRCLM_CAT_CD,
 ''20'' AS L2200CX_STC0102_HTCRCLM_STATCD,
 ''WQ'' AS L2200CX_STC03_ACTN_CD,
 BLLR.BLLR_CLM_CHG_AMT::FLOAT AS L2200CX_STC04_TOT_SBR_CHG_WRK,

BLLR.BLLR_CLM_CT::FLOAT AS L2200C_QTY02_QTY_APPR,
 BLLR.BLLR_CLM_CHG_AMT::FLOAT AS L2200C_AMT02_IN_PROCS,

MEM.LAST_NM AS L2100D_NM103_PT_LNM,
 MEM.FIRST_NM AS L2100D_NM104_PT_FNM,
 MEM.MEM_ID AS L2100D_NM109_MEM_ID_NR,

CLM_NR AS L2200D_TRN02_PT_CTL_NR,
 ''A1'' AS L2200DX_STC0101_HTCRCLM_CAT_CD,
 ''20'' AS L2200DX_STC0102_HTCRCLM_STATCD,

LOCALTIMESTAMP() AS L2200DX_STC02_STMT_NFO_EFF_D8,
 ''WQ'' AS L2200DX_STC03_ACTN_CD,
 CLM.CLM_CHG_AMT AS L2200DX_STC04_TOT_CLM_CHG_AMT,

```

        CLM_DT                AS L2200D_DTP_SVC_D8,
        1                      AS NEWROW

FROM    VW_SAMPL_MEMBER MEM
INNER JOIN
        SAMPL_PROVIDER P1
ON      P1.PROVIDER_ID = MEM.PAYER_PROVIDER_ID

INNER JOIN
        SAMPL_PROVIDER P2
ON      P2.PROVIDER_ID = MEM.BILLER_PROVIDER_ID

INNER JOIN
(
    SELECT  COUNT(*)                AS BLLR_CLM_CT,
            SUM(CHG_AMT)::FLOAT    AS BLLR_CLM_CHG_AMT,
            MEM2.BILLER_PROVIDER_ID
    FROM    SAMPL_CLAIM C
    INNER JOIN SAMPL_CLAIM_DTL D
    ON      C.CLAIM_ID = D.CLAIM_ID
    INNER JOIN VW_SAMPL_MEMBER MEM2
    ON      MEM2.MEMBER_ID = C.MEMBER_ID
    GROUP BY MEM2.BILLER_PROVIDER_ID

) BLLR
ON      BLLR.BILLER_PROVIDER_ID = P2.PROVIDER_ID

INNER JOIN
(
    SELECT  C.CLM_NR,
            C.MEMBER_ID,
            MIN(D.SVC_DT)           AS CLM_DT,
            SUM(CHG_AMT)::FLOAT    AS CLM_CHG_AMT
    FROM    SAMPL_CLAIM C
    INNER JOIN SAMPL_CLAIM_DTL D
    ON      C.CLAIM_ID = D.CLAIM_ID
    GROUP BY C.CLM_NR,
            C.MEMBER_ID

) CLM ON CLM.MEMBER_ID = MEM.MEMBER_ID

INNER JOIN
        SAMPL_HEADER SH
ON      SH.HDR_NAME = ''XPT_277CA''

ORDER BY
        P2.PROVIDER_ID,
        P1.PROVIDER_ID,
        CLM.CLM_NR
);

RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_278_REQ_EXTRACT()
RETURNS TABLE ();

```

```

LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN

```

```

-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

```

```

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_278_REQ';

```

```

-- Extraction

```

```

LET result RESULTSET := (
  SELECT  ISA_ISA02_NO_AUTH_NFO          AS Q0_ISA_ISA02_NO_AUTH_NFO,
          ISA_ISA04_PSSWD,
          ISA_ISA06_MUTLY_DEF,
          ISA_ISA08_MUTLY_DEF,
          ISA_ISA11_REPTN_SEP,
          ISA_ISA12_ICN_VERS_NR,
          ISA_ISA13_ICN,
          ISA_ISA15_ICN_USG_IND,
          ISA_ISA16_COMP_ELE_SEP,
          ISA_ISA13_ICN                  AS ISA_IEA02_ICN,
          GSHDR_GS02_APP_SNDR_CD,
          GSHDR_GS03_APP_RCV_CD,
          GSHDR_GS06_GCN,
          GSHDR_GS06_GCN                AS GSHDR_GE02_GCN,
          (100000000 + MEM.MEMBER_ID)::VARCHAR
          AS STHDR_ST02_TCN,
          '13'                          AS STHDR_BHT02_TS_PURP_CD,
          (100000000 + MEM.BILLER_PROVIDER_ID)::VARCHAR
          AS STHDR_BHT03_SBM_TRANS_ID,
          CURRENT_TIMESTAMP()           AS STHDR_BHT04_TS_CRTN_D8,
          CURRENT_TIMESTAMP()           AS STHDR_BHT05_TS_CRTN_TM,
          (100000000 + MEM.MEMBER_ID)::VARCHAR
          AS STHDR_SE02_TCN,
          '2'                            AS L2010A_PR_NM102_ENT_TYP_QUAL,
          P1.PROV_ORG_NM                 AS L2010A_PR_NM103_UMO_LNM,
          P1.TAX_ID                     AS L2010A_PR_NM109_EMPLYR_ID,
          '2'                            AS L2010B_2B_NM102_ENT_TYP_QUAL,
          P2.PROV_ORG_NM                 AS L2010B_2B_NM103_REQ_LNM,
          P2.PROV_NPI                   AS L2010B_2B_NM109_NPI,
          MEM.LAST_NM                   AS L2010C_NM103_SBR_LNM,
          MEM.FIRST_NM                  AS L2010C_NM104_SBR_FNM,
          MEM.MEM_ID                    AS L2010C_NM109_MEM_ID_NR,
          'HS'                          AS L2000E_UM01_REQST_CAT_CD,
          'I'                            AS L2000E_UM02_CERT_TYP_CD,
          CLM.POS_CD                    AS L2000E_UM0401_FAC_TYP_CD,
          'B'                            AS L2000E_UM0402_FAC_CD_QUAL,
          'Y'                            AS L2000E_UM09_RELS_NFO_CD,
          CLM.CLM_DT                    AS L2000E_DTP_ADMSN_D8,
          'HS'                          AS L2000F_UM01_REQST_CAT_CD,
          '1'                            AS L2000F_UM03_SVC_TYP_CD,
          SVC.SVC_DT                    AS L2000F_DTP_SVC_D8,
          SVC.HCPCS_CD                  AS L2000F_SV10102_HCPCS_CD,
          SVC.MOD01                     AS L2000F_SV10103_PROC_MOD,
          SVC.MOD02                     AS L2000F_SV10104_PROC_MOD,
          SVC.UNITS::FLOAT              AS L2000F_SV104_UN,
          '2'                            AS L2010F_QB_NM102_ENT_TYP_QUAL,
          P2.PROV_ORG_NM                 AS L2010F_QB_NM103_SVC_PVR_LNM,
          P2.PROV_NPI                   AS L2010F_QB_NM109_NPI,
          1                              AS NEWROW
  FROM    VW_SAMPL_MEMBER MEM
  INNER JOIN
          SAMPL_PROVIDER P1

```

```

ON      P1.PROVIDER_ID = MEM.PAYER_PROVIDER_ID
INNER JOIN
      SAMPL_PROVIDER P2
ON      P2.PROVIDER_ID = MEM.BILLER_PROVIDER_ID
INNER JOIN
(
      SELECT  C.CLAIM_ID,
              C.CLM_NR,
              C.POS_CD,
              C.MEMBER_ID,
              MIN(D.SVC_DT)           AS CLM_DT,
              SUM(CHG_AMT)::FLOAT     AS CLM_CHG_AMT
      FROM    SAMPL_CLAIM C
      INNER JOIN SAMPL_CLAIM_DTL D
      ON      C.CLAIM_ID = D.CLAIM_ID
      GROUP BY C.CLAIM_ID,
              C.POS_CD,
              C.CLM_NR,
              C.MEMBER_ID

) CLM ON CLM.MEMBER_ID = MEM.MEMBER_ID
INNER JOIN
      SAMPL_CLAIM_DTL SVC
ON      SVC.CLAIM_ID = CLM.CLAIM_ID

INNER JOIN
      SAMPL_HEADER SH
ON      SH.HDR_NAME = 'XPT_278_REQ'
WHERE   CLM.POS_CD <> '13'
ORDER BY
      MEM.MEMBER_ID
);

RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_278_RESP_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----
-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_278_RESP';

-- Extraction
LET result RESULTSET := (
      SELECT  ISA_ISA02_NO_AUTH_NFO           AS R0_ISA_ISA02_NO_AUTH_NFO,
              ISA_ISA04_PSSWD,
              ISA_ISA06_MUTLY_DEF,
              ISA_ISA08_MUTLY_DEF,
              ISA_ISA11_REPTN_SEP,
              ISA_ISA12_ICN_VERS_NR,
              ISA_ISA13_ICN,
              ISA_ISA15_ICN_USG_IND,

```

```

ISA_ISA16_COMP_ELE_SEP,
ISA_ISA13_ICN          AS ISA_IEA02_ICN,
GSHDR_GS02_APP_SNDR_CD,
GSHDR_GS03_APP_RCV_CD,
GSHDR_GS06_GCN,
GSHDR_GS06_GCN          AS GSHDR_GE02_GCN,
(100000000 + MEM.MEMBER_ID)::VARCHAR
                        AS STHDR_ST02_TCN,
(100000000 + MEM.BILLER_PROVIDER_ID)::VARCHAR
                        AS STHDR_BHT03_SBM_TRANS_ID,
CURRENT_TIMESTAMP()    AS STHDR_BHT04_TS_CRTN_D8,
CURRENT_TIMESTAMP()    AS STHDR_BHT05_TS_CRTN_TM,
''18''                  AS STHDR_BHT06_TRANS_TYP_CD,
(100000000 + MEM.MEMBER_ID)::VARCHAR
                        AS STHDR_SE02_TCN,
''2''                  AS L2010A_2B_NM102_ENT_TYP_QUAL,
P2.PROV_ORG_NM          AS L2010A_2B_NM103_UMO_LNM,
P2.TAX_ID              AS L2010A_2B_NM109_EMPLYR_ID,
''2''                  AS L2010B_1P_NM102_ENT_TYP_QUAL,
P1.PROV_ORG_NM          AS L2010B_1P_NM103_REQ_LNM,
P1.PROV_NPI            AS L2010B_1P_NM109_NPI,
MEM.LAST_NM            AS L2010C_NM103_SBR_LNM,
MEM.FIRST_NM           AS L2010C_NM104_SBR_FNM,
MEM.MEM_ID             AS L2010C_NM109_MEM_ID_NR,
MEM.DOB                AS L2010C_DMG02_D8,
MEM.GENDER             AS L2010C_DMG03_SUB_GENDR_CD,
CLM.CLM_NR            AS L2010C_REF_PATNT_ACCT_NR,
''A1''                 AS L2000E_HCR01_ACTN_CD,
(100000000 + MEM.BILLER_PROVIDER_ID)::VARCHAR
                        AS L2000E_HCR02_RVW_ID_NR,
''2''                  AS L2000E_01TRN01_TRAC_TYP_CD,
(100000000 + MEM.BILLER_PROVIDER_ID)::VARCHAR
                        AS L2000E_01TRN02_PTEVT_TRAC_NR,
P2.TAX_ID              AS L2000E_01TRN03_TRAC_ASS_ID,
''HS''                 AS L2000E_UM01_REQST_CAT_CD,
''I''                  AS L2000E_UM02_CERT_TYP_CD,
CLM.POS_CD            AS L2000E_UM0401_FAC_TYP_CD,
''B''                  AS L2000E_UM0402_FAC_CD_QUAL,
1                      AS NEWROW
FROM VW_SAMPL_MEMBER MEM
INNER JOIN
    SAMPL_PROVIDER P1
ON P1.PROVIDER_ID = MEM.PAYER_PROVIDER_ID
INNER JOIN
    SAMPL_PROVIDER P2
ON P2.PROVIDER_ID = MEM.BILLER_PROVIDER_ID
INNER JOIN
    SAMPL_CLAIM CLM
ON CLM.MEMBER_ID = MEM.MEMBER_ID
INNER JOIN
    SAMPL_HEADER SH
ON SH.HDR_NAME = 'XPT_278_RESP'
WHERE CLM.POS_CD <> '13'
ORDER BY
    MEM.MEMBER_ID
);

RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_820_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----
-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_820';

-- Extraction
LET result RESULTSET := (
    SELECT  ISA_ISA02_NO_AUTH_NFO          AS S0_ISA_ISA02_NO_AUTH_NFO,
            ISA_ISA04_PSSWD,
            ISA_ISA06_MUTLY_DEF,
            ISA_ISA08_MUTLY_DEF,
            ISA_ISA11_REPTN_SEP,
            ISA_ISA12_ICN_VERS_NR,
            ISA_ISA13_ICN,
            ISA_ISA15_ICN_USG_IND,
            ISA_ISA16_COMP_ELE_SEP,
            ISA_ISA13_ICN                  AS ISA_IEA02_ICN,
            GSHDR_GS02_APP_SNDR_CD,
            GSHDR_GS03_APP_RCV_CD,
            GSHDR_GS06_GCN,
            GSHDR_GS06_GCN                AS GSHDR_GE02_GCN,
            (10000000 + PMT.P1)::VARCHAR
            AS STHDR_ST02_TCN,
            'I'                            AS STHDR_BPR01_TRANS_HANDL_CD,
            (PMT.CT * 10.00)::FLOAT        AS STHDR_BPR02_TOT_PRM_PMT_AMT,
            'CHK'                          AS STHDR_BPR04_PMT_METHD_CD,
            ('1' || P1.TAX_ID)             AS STHDR_BPR10_PYR_ID,
            CURRENT_TIMESTAMP()            AS STHDR_BPR16_CHK_EFT_EFDT_D8,
            '3'                            AS STHDR_TRN01_TRAC_TYP_CD,
            ('CHK000' || P1.PROVIDER_ID)::VARCHAR
            AS STHDR_TRN02_CHK_EFT_TRC_NR,
            (10000000 + PMT.P1)::VARCHAR
            AS STHDR_SE02_TCN,
            P1.PROV_ORG_NM                  AS L1000A_N102_PREM_RCV_LNM,
            P1.TAX_ID                      AS L1000A_N104_TAX_ID,
            P2.PROV_ORG_NM                  AS L1000B_N102_PREM_PYR_NM,
            P2.TAX_ID                      AS L1000B_N104_TAX_ID,
            ROW_NUMBER() OVER (PARTITION BY P2.PROVIDER_ID, P1.PROVIDER_ID ORDER BY P2.PROVIDER_ID,
P1.PROVIDER_ID, M1.MEM_ID)
            AS L2000B_ENT01_ASSGD_NR,
            M1.SSN                        AS L2000B_ENT04_SSN,
            M1.LAST_NM                    AS L2100BX_NM103_IND_LNM,
            M1.FIRST_NM                   AS L2100BX_NM104_IND_FNM,
            M1.MEM_ID                     AS L2100BX_NM109_INSRD_UNQ_ID,
            ('INV' || M1.MEMBER_ID)::VARCHAR
            AS L2300B_RMR02_INVC_NR,
            10.0::FLOAT                   AS L2300B_RMR04_DTL_PRM_PMT_AMT,
            1                             AS NEWROW
    FROM    SAMPL_HEADER SH
    INNER JOIN VW_SAMPL_MEMBER M1
    ON      1=1
);

```

```

INNER JOIN SAMPL_PROVIDER P1
ON P1.PROVIDER_ID = M1.PAYER_PROVIDER_ID
INNER JOIN SAMPL_PROVIDER P2
ON P2.PROVIDER_ID = M1.BILLER_PROVIDER_ID
INNER JOIN
(
    SELECT COUNT(*)::FLOAT AS CT,
           P1.PROVIDER_ID AS P1,
           P2.PROVIDER_ID AS P2
    FROM VW_SAMPL_MEMBER V
    INNER JOIN SAMPL_PROVIDER P1
    ON V.BILLER_PROVIDER_ID = P1.PROVIDER_ID
    INNER JOIN SAMPL_PROVIDER P2
    ON V.PAYER_PROVIDER_ID = P2.PROVIDER_ID
    GROUP BY P1.PROVIDER_ID, P2.PROVIDER_ID
) PMT ON PMT.P2 = M1.PAYER_PROVIDER_ID AND PMT.P1 = M1.BILLER_PROVIDER_ID
WHERE SH.HDR_NAME = 'XPT_820'
ORDER BY P2.PROVIDER_ID,
         P1.PROVIDER_ID,
         M1.MEM_ID
);

```

```

RETURN TABLE(result);
END;
';

```

USP_824_EXTRACT

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

```

CREATE OR REPLACE PROCEDURE MAIN.USP_824_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----
-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_824';

-- Extraction
LET result RESULTSET := (
    SELECT ISA_ISA02_NO_AUTH_NFO AS P7_ISA_ISA02_NO_AUTH_NFO,
           ISA_ISA04_PSSWD,
           ISA_ISA06_MUTLY_DEF,
           ISA_ISA08_MUTLY_DEF,
           ISA_ISA11_REPTN_SEP,
           ISA_ISA12_ICN_VERS_NR,
           ISA_ISA13_ICN,
           ISA_ISA15_ICN_USG_IND,
           ISA_ISA16_COMP_ELE_SEP,
           ISA_ISA13_ICN AS ISA_IEA02_ICN,
           GSHDR_GS02_APP_SNDR_CD,
           GSHDR_GS03_APP_RCV_CD,
           GSHDR_GS06_GCN,
           GSHDR_GS06_GCN AS GSHDR_GE02_GCN,
           '100000000' AS STHDR_ST02_TCN,
           '11111' AS STHDR_BGN02_TS_ID_CD,
           CURRENT_TIMESTAMP() AS STHDR_BGN03_TS_CRTN_D8,
           CURRENT_TIMESTAMP() AS STHDR_BGN04_TS_CRTN_TM,
           '1234567890' AS STHDR_BGN06_REF_ICN,
           'WQ' AS STHDR_BGN08_ACTN_CD,

```

```

        ''10000000''          AS STHDR_SE02_TCN,
P1.PROV_ORG_NM              AS L1000A_N102_SBM_NM,
P1.TAX_ID                   AS L1000A_N104_ETIN,
''TECH SUPPORT''           AS L1000A_PER02_SBM_CON_NM,
''4155551212''             AS L1000A_PER04_PHN_NR,
P2.PROV_ORG_NM              AS L1000B_N102_RCV_NM,
P2.TAX_ID                   AS L1000B_N104_ETIN,
''TA''                      AS L2000_OTI01_APP_ACK_CD,
''10000000''               AS L2000_OTI03_TRAN_REF_NR,
''20180101'':::DATE        AS L2000_OTI06_FUNC_GRP_CRTN_D8,
''1900-01-01 05:00:00'':::TIMESTAMP_NTZ
                                AS L2000_OTI07_FUNC_GRP_CRTN_TM,
                                AS L2000_OTI08_FUNC_GRP_CTL_NR,
                                AS L2000_OTI09_TCN,
                                AS L2000_OTI10_TS_ID_CD,
                                AS L2000_OTI11_VERS_RLS_IND_ID,
                                AS NEWROW
1
''10000000''
''834''
''005010X220A1''
1
FROM    SAMPL_HEADER
INNER JOIN SAMPL_PROVIDER P1
ON      P1.PROVIDER_ID = 1
INNER JOIN SAMPL_PROVIDER P2
ON      P2.PROVIDER_ID = 2
WHERE   HDR_NAME = ''XPT_824''
);

RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_834_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN
-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----
-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = ''XPT_834'';

-- Extraction
LET result RESULTSET := (
    SELECT  ISA_ISA02_NO_AUTH_NFO          AS T1_ISA_ISA02_NO_AUTH_NFO,
            ISA_ISA04_PSSWD,
            ISA_ISA06_MUTLY_DEF,
            ISA_ISA08_MUTLY_DEF,
            ISA_ISA11_REPTN_SEP,
            ISA_ISA12_ICN_VERS_NR,
            ISA_ISA13_ICN,
            ISA_ISA15_ICN_USG_IND,
            ISA_ISA16_COMP_ELE_SEP,
            ISA_ISA13_ICN                  AS ISA_IEA02_ICN,
            GSHDR_GS02_APP_SNDR_CD,
            GSHDR_GS03_APP_RCV_CD,
            GSHDR_GS06_GCN,
            GSHDR_GS06_GCN                AS GSHDR_GE02_GCN,

```

```

(10000000 + MEM.BILLER_PROVIDER_ID)::VARCHAR
      AS STHDR_ST02_TCN,
''00''
      AS STHDR_BGN01_TS_PURP_CD,
''0001''
      AS STHDR_BGN02_TS_REF_NR,
CURRENT_TIMESTAMP()::TIMESTAMP_NTZ
      AS STHDR_BGN03_TS_CRTN_D8,
CURRENT_TIMESTAMP()::TIMESTAMP_NTZ
      AS STHDR_BGN04_TS_CRTN_TM,
''4''
      AS STHDR_BGN08_ACTN_CD,
(10000000 + MEM.BILLER_PROVIDER_ID)::VARCHAR
      AS STHDR_SE02_TCN,
P1.PROV_ORG_NM
      AS L1000A_N102_PLN_SPNSR_NM,
P1.TAX_ID
      AS L1000A_N104_TAX_ID,
P2.PROV_ORG_NM
      AS L1000B_N102_INSR_NM,
P2.TAX_ID
      AS L1000B_N104_TAX_ID,
IS_SUBSCRIBER
      AS L2000_INS01_MEM_IND,
CASE WHEN IS_SUBSCRIBER = 'Y' THEN '18' ELSE '19' END
      AS L2000_INS02_IND_RELAT_CD,
''030''
      AS L2000_INS03_MAINT_TYP_CD,
''XN''
      AS L2000_INS04_MAINT_RSN_CD,
''A''
      AS L2000_INS05_BENE_STAT_CD,
CASE WHEN DATEADD(YEAR, 65, DOB) < CURRENT_TIMESTAMP()::TIMESTAMP_NTZ THEN 'E' ELSE NULL
      AS L2000_INS0601_MDCR_PLAN_CD,
CASE WHEN IS_SUBSCRIBER = 'Y' THEN 'AC' ELSE NULL END
      AS L2000_INS08_EMPMT_STAT_CD,
MEM_ID
      AS L2000_REF_SUB_NR,
''POLCY001''
      AS L2000_REF_GRP_POLCY_NR,
''2020-01-01''::TIMESTAMP_NTZ
      AS L2000_DTP_ELIG_BGN_D8,
LAST_NM
      AS L2100A_IL_NM103_PERSN_LNM,
FIRST_NM
      AS L2100A_IL_NM104_MBR_FNM,
SSN
      AS L2100A_IL_NM109_SSN,
CASE WHEN IS_SUBSCRIBER = 'Y' THEN RES_ADDR ELSE NULL END
      AS L2100A_IL_N301_MBR_ADDR,
CASE WHEN IS_SUBSCRIBER = 'Y' THEN RES_CITY ELSE NULL END
      AS L2100A_IL_N401_MBR_CITY,
CASE WHEN IS_SUBSCRIBER = 'Y' THEN RES_STATE ELSE NULL END
      AS L2100A_IL_N402_MBR_STAT,
CASE WHEN IS_SUBSCRIBER = 'Y' THEN RES_ZIP ELSE NULL END
      AS L2100A_IL_N403_MBR_ZIP,
DOB
      AS L2100A_IL_DMG02_D8,
GENDER
      AS L2100A_IL_DMG03_GENDR_CD,
''030''
      AS L2300_HD01_MAINT_TYP_CD,
''HMO''
      AS L2300_HD03_INS_LIN_CD,
''2020-01-01''::TIMESTAMP_NTZ
      AS L2300_DTP_BNFT_BGN_D8,
''2020-12-31''::TIMESTAMP_NTZ
      AS L2300_02DTP_BNFT_END_D8,
1
      AS NEWROW
FROM VW_SAMPL_MEMBER MEM
INNER JOIN
      SAMPL_PROVIDER P1
ON P1.PROVIDER_ID = MEM.PAYER_PROVIDER_ID
INNER JOIN
      SAMPL_PROVIDER P2
ON P2.PROVIDER_ID = MEM.BILLER_PROVIDER_ID
INNER JOIN
      SAMPL_HEADER SH
ON SH.HDR_NAME = 'XPT_834'
ORDER BY
      MEM.BILLER_PROVIDER_ID,
      MEM.PAYER_PROVIDER_ID,
      MEM.MEMBER_ID DESC
);

RETURN TABLE(result);
END;
';

```

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.

```
CREATE OR REPLACE PROCEDURE MAIN.USP_835_EXTRACT()  
RETURNS TABLE ()  
LANGUAGE SQL  
EXECUTE AS OWNER  
AS '  
BEGIN
```

```
-----  
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.  
-----
```

```
-- Increment Interchange Control Number
```

```

UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_835';

```

```
-- Extraction
```

```

LET result RESULTSET := (
  SELECT  ISA_ISA02_NO_AUTH_NFO      AS U1_ISA_ISA02_NO_AUTH_NFO,
          ISA_ISA04_PSSWD,
          ISA_ISA06_MUTLY_DEF,
          ISA_ISA08_MUTLY_DEF,
          ISA_ISA11_REPTN_SEP,
          ISA_ISA12_ICN_VERS_NR,
          ISA_ISA13_ICN,
          ISA_ISA15_ICN_USG_IND,
          ISA_ISA16_COMP_ELE_SEP,
          ISA_ISA13_ICN              AS ISA_IEA02_ICN,
          GSHDR_GS02_APP_SNDR_CD,    AS GSHDR_GE02_GCN,
          GSHDR_GS03_APP_RCV_CD,
          GSHDR_GS06_GCN,
          GSHDR_GS06_GCN              AS GSHDR_GE02_GCN,

          (10000000 + P1.PROVIDER_ID)::VARCHAR
          AS STHDR_ST02_TCN,
          'I'                          AS STHDR_BPR01_TRANS_HANDL_CD,
          CHK_PMT_AMT::FLOAT           AS STHDR_BPR02_TOT_ACTL_PVR_PMT,
          'C'                          AS STHDR_BPR03_CRED_DEB_FLG_CD,
          'CHK'                        AS STHDR_BPR04_PMT_METHD_CD,
          CURRENT_TIMESTAMP()::TIMESTAMP_NTZ
          AS STHDR_BPR16_CHK_EFT_EFDT_D8,
          'CHK00001'                  AS STHDR_TRN02_CHK_EFT_TRC_NR,
          P1.TAX_ID                    AS STHDR_TRN03_PYR_ID,
          (10000000 + P1.PROVIDER_ID)::VARCHAR
          AS STHDR_SE02_TCN,

          P1.PROV_ORG_NM                AS L1000A_N102_PYR_NM,
          P1.BIZ_ADDR                   AS L1000A_N301_PYR_ADDR_LN,
          P1.BIZ_CITY                   AS L1000A_N401_PYR_CITY_NM,
          P1.BIZ_STATE                  AS L1000A_N402_PYR_STAT,
          P1.BIZ_ZIP                    AS L1000A_N403_PYR_ZIP,
          P1.TAX_ID                     AS L1000A_REF_PYR_ID,
          'TECH CONTACT'                AS L1000A_PERB02_PYR_CON_NM,
          P1.BIZ_PHONE                  AS L1000A_PERB04_PHN_NR,

          P2.PROV_ORG_NM                AS L1000B_N102_PAYEE_NM,
          P2.PROV_NPI                   AS L1000B_N104_NPI,
          P2.BIZ_ADDR                   AS L1000B_N301_PAYEE_ADDR,
          P2.BIZ_CITY                   AS L1000B_N401_PAYEE_CITY,
          P2.BIZ_STATE                  AS L1000B_N402_PAYEE_STAT,
          P2.BIZ_ZIP                    AS L1000B_N403_PAYEE_ZIP,
          P2.TAX_ID                     AS L1000B_REF_TAX_ID,

          CLM.PT_CTL_NR                 AS L2100_CLP01_PT_CTL_NR,
          '1'                            AS L2100_CLP02_CLM_STAT_CD,
          CLM_NFO.CLM_CHG_AMT::FLOAT    AS L2100_CLP03_TOT_CLM_CHG_AMT,
          CLM_NFO.CLM_PMT_AMT::FLOAT    AS L2100_CLP04_CLM_PMT_AMT,
          CASE WHEN CLM_NFO.CLM_PR_AMT = 0
                THEN NULL
                ELSE CLM_NFO.CLM_PR_AMT::FLOAT

          AS L2100_CLP05_PT_RESP_AMT,
          'HM'                          AS L2100_CLP06_CLM_FIL_IND_CD,
          CLM_NR                         AS L2100_CLP07_PYR_CLM_CTL_NR,
          POS_CD                         AS L2100_CLP08_FAC_TYP_CD,
          M1.LAST_NM                     AS L2100_NM1B03_PT_LNM,
          M1.FIRST_NM                    AS L2100_NM1B04_PT_FNM,
          M1.MEM_ID                      AS L2100_NM1B09_MEM_ID_NR,
          'POLCY0001'                   AS L2100_REF_GRP_POLCY_NR,

```

```

        ''C001''                AS L2100_02REF_CLSS_CONT,
        CLM_BEGIN                AS L2100_DTM02_CLM_PRD_STRT_D8,
        CLM_END                  AS L2100_02DTM02_CLM_PRD_END_D8,
        CLM_CHG_AMT::FLOAT       AS L2100_AMT02_COVG_AMT,

        HCPCS_CD                 AS L2110_SVC0102_HCPCS_CD,
        MOD01                     AS L2110_SVC0103_PROC_MOD,
        MOD02                     AS L2110_SVC0104_PROC_MOD,
        CHG_AMT::FLOAT           AS L2110_SVC02_LIN_ITM_CHG_AMT,
        PMT_AMT::FLOAT           AS L2110_SVC03_LIN_ITM_PV_PMT_AMT,
        UNITS::FLOAT             AS L2110_SVC05_UN_SVC_PD_CT,
        SVC_DT                    AS L2110_DTM02_SVC_D8,
        CHG_AMT::FLOAT           AS L2110_AMT02_ALLWD_ACTL,

        L2110X_CAS01_CLMADJ_GRP_CD,
        L2110X_CAS02_ADJ_RSN_CD,
        L2110X_CAS03_ADJ_AMT,
        L2110X_CAS05_ADJ_RSN_CD,
        L2110X_CAS06_ADJ_AMT,
        L2110X_CAS08_ADJ_RSN_CD,
        L2110X_CAS09_ADJ_AMT,
        NEWROW

FROM    SAMPL_HEADER SH

INNER JOIN VW_SAMPL_MEMBER M1
ON      1=1

INNER JOIN SAMPL_PROVIDER P1
ON      P1.PROVIDER_ID = M1.PAYER_PROVIDER_ID

INNER JOIN SAMPL_PROVIDER P2
ON      P2.PROVIDER_ID = M1.BILLER_PROVIDER_ID

INNER JOIN SAMPL_CLAIM CLM
ON      CLM.MEMBER_ID = M1.MEMBER_ID

INNER JOIN SAMPL_CLAIM_DTL DTL
ON      DTL.CLAIM_ID = CLM.CLAIM_ID

INNER JOIN
(
    SELECT  SUM(PMT_AMT)           AS CHK_PMT_AMT,
            PAYER_PROVIDER_ID,
            BILLER_PROVIDER_ID
    FROM    SAMPL_CLAIM_DTL D
    INNER JOIN SAMPL_CLAIM C
    ON      D.CLAIM_ID = C.CLAIM_ID
    INNER JOIN VW_SAMPL_MEMBER M
    ON      C.MEMBER_ID = M.MEMBER_ID
    GROUP BY M.PAYER_PROVIDER_ID,
            M.BILLER_PROVIDER_ID
) CHK_NFO
ON      CHK_NFO.PAYER_PROVIDER_ID = P1.PROVIDER_ID

INNER JOIN
(
    SELECT  D.CLAIM_ID,
            SUM(PMT_AMT)           AS CLM_PMT_AMT,
            SUM(CHG_AMT)           AS CLM_CHG_AMT,
            SUM(COALESCE(COPAY, 0.00) + COALESCE(COINS, 0.00) + COALESCE(DEDUCTIBLE, 0.00)) AS
CLM_PR_AMT,
            MIN(SVC_DT)            AS CLM_BEGIN,
            MAX(SVC_DT)            AS CLM_END,
            PAYER_PROVIDER_ID
    FROM    SAMPL_CLAIM_DTL D

```

```

INNER JOIN SAMPL_CLAIM C
ON      D.CLAIM_ID = C.CLAIM_ID
INNER JOIN VW_SAMPL_MEMBER M
ON      C.MEMBER_ID = M.MEMBER_ID
GROUP BY D.CLAIM_ID,
        M.PAYER_PROVIDER_ID
) CLM_NFO
ON      CLM_NFO.CLAIM_ID = CLM.CLAIM_ID

INNER JOIN
(
    -- CO*45 WITH OR WITHOUT WITHHOLDINGS, NEWROW = 1 IF NO PR, 0 IF PR
    SELECT CLAIM_DTL_ID,
           'CO' AS L2110X_CAS01_CLMADJ_GRP_CD,
           CASE WHEN COALESCE(WITHHOLD, 0.00) > 0.00
                THEN '104'
                ELSE '45'
           END AS L2110X_CAS02_ADJ_RSN_CD,
           CASE WHEN COALESCE(WITHHOLD, 0.00) > 0.00
                THEN WITHHOLD::FLOAT
                ELSE (CHG_AMT - PMT_AMT - COALESCE(COPAY, 0.00) - COALESCE(DEDUCTIBLE, 0.00)
           - COALESCE(COINS, 0.00) - COALESCE(WITHHOLD, 0.00))::FLOAT
           END AS L2110X_CAS03_ADJ_AMT,
           CASE WHEN COALESCE(WITHHOLD, 0.00) > 0.00
                THEN '45'
                ELSE NULL
           END AS L2110X_CAS05_ADJ_RSN_CD,
           CASE WHEN COALESCE(WITHHOLD, 0.00) > 0.00
                THEN (CHG_AMT - PMT_AMT - COALESCE(COPAY, 0.00) - COALESCE(DEDUCTIBLE, 0.00)
           - COALESCE(COINS, 0.00) - COALESCE(WITHHOLD, 0.00))::FLOAT
                ELSE NULL
           END AS L2110X_CAS06_ADJ_AMT,
           NULL AS L2110X_CAS08_ADJ_RSN_CD,
           NULL AS L2110X_CAS09_ADJ_AMT,
           CASE WHEN COALESCE(COPAY, 0.00) + COALESCE(COINS, 0.00) + COALESCE(DEDUCTIBLE,
0.00) = 0.00
                THEN 1
                ELSE 0
           END AS NEWROW
    FROM    SAMPL_CLAIM_DTL D

    UNION

    -- PR > 0, SO CAS*PR SEGMENT EMITTED, NEWROW = 1
    SELECT CLAIM_DTL_ID,
           'PR' AS L2110X_CAS01_CLMADJ_GRP_CD,
           CASE WHEN COALESCE(DEDUCTIBLE, 0.00) > 0.00
                THEN '1'
                ELSE NULL
           END AS L2110X_CAS02_ADJ_RSN_CD,
           DEDUCTIBLE::FLOAT AS L2110X_CAS03_ADJ_AMT,
           CASE WHEN COALESCE(COINS, 0.00) > 0.00
                THEN '2'
                ELSE NULL
           END AS L2110X_CAS05_ADJ_RSN_CD,
           COINS::FLOAT AS L2110X_CAS06_ADJ_AMT,
           CASE WHEN COALESCE(COPAY, 0.00) > 0.00
                THEN '3'
                ELSE NULL
           END AS L2110X_CAS08_ADJ_RSN_CD,
           COPAY::FLOAT AS L2110X_CAS09_ADJ_AMT,
           1 AS NEWROW
    FROM    SAMPL_CLAIM_DTL D
    WHERE   COALESCE(COPAY, 0.00) + COALESCE(COINS, 0.00) + COALESCE(DEDUCTIBLE, 0.00) >
0.00
) CAS_DTL

```

```

        ON      CAS_DTL.CLAIM_DTL_ID = DTL.CLAIM_DTL_ID

        WHERE   SH.HDR_NAME = 'XPT_835'

        ORDER BY P1.PROVIDER_ID,
                CLM.CLAIM_ID,
                CAS_DTL.CLAIM_DTL_ID,
                CAS_DTL.NEWROW DESC
    );

    RETURN TABLE(result);
END;
';

```

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.

```

CREATE OR REPLACE PROCEDURE MAIN.USP_837I_EXTRACT()
RETURNS TABLE ()

```

```

LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN

```

```

-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

```

```

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_837I';

```

```

-- Extraction

```

```

LET result RESULTSET := (
  SELECT  ISA_ISA02_NO_AUTH_NFO      AS W2_ISA_ISA02_NO_AUTH_NFO,
          ISA_ISA04_PSSWD,
          ISA_ISA06_MUTLY_DEF,
          ISA_ISA08_MUTLY_DEF,
          ISA_ISA11_REPTN_SEP,
          ISA_ISA12_ICN_VERS_NR,
          ISA_ISA13_ICN,
          ISA_ISA15_ICN_USG_IND,
          ISA_ISA16_COMP_ELE_SEP,
          ISA_ISA13_ICN              AS ISA_IEA02_ICN,

          GSHDR_GS02_APP_SNDR_CD,
          GSHDR_GS03_APP_RCV_CD,
          GSHDR_GS06_GCN,
          GSHDR_GS06_GCN            AS GSHDR_GE02_GCN,

          (100000000 + MEM.SUB_BILLER_PROVIDER_ID)::VARCHAR
          AS STHDR_ST02_TCN,
          '0019'                    AS STHDR_BHT01_HIER_STRUC_CD,
          '00'                      AS STHDR_BHT02_TS_PURP_CD,
          'BAT1000000'              AS STHDR_BHT03_ORIG_APP_TRANS_ID,
          CURRENT_TIMESTAMP()::TIMESTAMP_NTZ
          AS STHDR_BHT04_TS_CRTN_D8,
          CURRENT_TIMESTAMP()::TIMESTAMP_NTZ
          AS STHDR_BHT05_TS_CRTN_TM,
          'CH'                      AS STHDR_BHT06_CLM_ID,
          (100000000 + MEM.SUB_BILLER_PROVIDER_ID)::VARCHAR
          AS STHDR_SE02_TCN,

          P2.PROV_ORG_NM            AS L1000A_NM103_NONPSNENT_NM,
          P2.TAX_ID                AS L1000A_NM109_ETN_NR,
          'SAMPLE'                 AS L1000A_PER02_SBM_CON_NM,
          '4155551212'            AS L1000A_PER04_PHN_NR,

          P1.PROV_ORG_NM            AS L1000B_NM103_NONPSNENT_NM,
          P1.PROV_NPI              AS L1000B_NM109_ETN_NR,

          P2.PROV_ORG_NM            AS L2010AA_NM103_NONPSNENT_NM,
          P2.PROV_NPI              AS L2010AA_NM109_NPI,
          P2.BIZ_ADDR              AS L2010AA_N301_BILL_PROV_ADDR,
          P2.BIZ_CITY              AS L2010AA_N401_BILL_PVR_CITY,
          P2.BIZ_STATE             AS L2010AA_N402_BILL_PVR_STAT,
          (P2.BIZ_ZIP || '0001')   AS L2010AA_N403_BILL_PVR_ZIP,
          P2.TAX_ID                AS L2010AA_REF_EMPLR_ID_NR,

          'P'                      AS L2000B_SBR01_PYR_RESP_SEQ_NR,
          CASE WHEN MEM.PAT_MEMBER_ID IS NULL THEN '18' ELSE NULL END
          AS L2000B_SBR02_IND_RELAT_CD,
          'GRP0001'                AS L2000B_SBR04_SBR_GRP_NM,
          'HM'                     AS L2000B_SBR09_CLM_FIL_IND_CD,

```

```

MEM.SUB_LAST_NM          AS L2010BA_NM103_PERSN_LNM,
MEM.SUB_FIRST_NM        AS L2010BA_NM104_SBR_FNM,
MEM.SUB_ID              AS L2010BA_NM109_MEM_ID_NR,
MEM.SUB_ADDR            AS L2010BA_N301_SBR_ADDR,
MEM.SUB_CITY            AS L2010BA_N401_SBR_CITY,
MEM.SUB_STATE           AS L2010BA_N402_SBR_STAT,
(MEM.SUB_ZIP || '0001') AS L2010BA_N403_SBR_ZIP,
MEM.SUB_DOB             AS L2010BA_DMG02_D8,
MEM.SUB_GENDER          AS L2010BA_DMG03_SUB_GENDR_CD,

P1.PROV_ORG_NM          AS L2010BB_NM103_NONPSNENT_NM,
P1.TAX_ID               AS L2010BB_NM109_PAYR_ID,
P1.BIZ_CITY             AS L2010BB_N401_PYR_CITY_NM,
P1.BIZ_STATE            AS L2010BB_N402_PYR_STAT,
(P1.BIZ_ZIP || '0001') AS L2010BB_N403_PYR_ZIP,

CASE WHEN MEM.PAT_LAST_NM IS NOT NULL THEN '19' ELSE NULL END
AS L2000C_PAT01_IND_RELAT_CD,

MEM.PAT_LAST_NM          AS L2010CA_NM103_PERSN_LNM,
MEM.PAT_FIRST_NM        AS L2010CA_NM104_PT_FNM,
MEM.PAT_ADDR            AS L2010CA_N301_PT_ADDR,
MEM.PAT_CITY            AS L2010CA_N401_PT_CITY,
MEM.PAT_STATE           AS L2010CA_N402_PT_STAT,
MEM.PAT_ZIP             AS L2010CA_N403_PT_ZIP,
MEM.PAT_DOB             AS L2010CA_DMG02_D8,
MEM.PAT_GENDER          AS L2010CA_DMG03_PAT_GNDR_CD,

CLM.PT_CTL_NR           AS L2300_CLM01_PT_CTL_NR,
CLM_CHG_AMT:::FLOAT     AS L2300_CLM02_TOT_CLM_CHG_AMT,
CLM.POS_CD              AS L2300_CLM0501_FAC_TYP_CD,
'1'                     AS L2300_CLM0503_CLM_FREQ_CD,
'A'                     AS L2300_CLM07_PLAN_PART_CD,
'Y'                     AS L2300_CLM08_BEN_ASGT_CRT_IND,
'Y'                     AS L2300_CLM09_RELS_NFO_CD,
'1'                     AS L2300_CL101_ADMSN_TYP_CD,
'1'                     AS L2300_CL102_ADMSN_SRC_CD,
'30'                    AS L2300_CL103_PT_STATS_CD,

CASE WHEN ADMIT_DIAG IS NOT NULL
      THEN CLM_NFO.CLM_BEGIN
      ELSE NULL

END
AS L2300_DTP_ADMSN_D8,
CLM_NFO.CLM_BEGIN       AS L2300_DTP_STMNT_RD8_1,
CLM_NFO.CLM_END         AS L2300_DTP_STMNT_RD8_2,
CLM.CLM_NR              AS L2300_REF_CLM_NR,
CLM.PRIN_DIAG           AS L2300_HI0102_ICD10_PRIN_DIAG,
CLM.ADMIT_DIAG          AS L2300_HI0102_ICD10_ADMTG_DIAG,
CLM.DIAG02              AS L2300_HIE0102_ICD10_DIAG,
CLM.DIAG03              AS L2300_HIE0202_ICD10_DIAG,

PRF.PROF_LNAME          AS L2310A_NM103_PERSN_LNM,
PRF.PROF_FNAME          AS L2310A_NM104_ATT_PVR_FNM,
PRF.PROF_NPI            AS L2310A_NM109_NPI,

DTL.LINE_SEQ            AS L2400_LX01_ASSGD_NR,
'0100'                  AS L2400_SV201_SVC_LIN_REV_CD,
DTL.HCPCS_CD            AS L2400_SV20202_HCPCS_CD,
DTL.DESCR               AS L2400_SV20207_DESCR,
DTL.CHG_AMT:::FLOAT     AS L2400_SV203_LIN_ITM_CHG_AMT,
DTL.UNITS:::FLOAT       AS L2400_SV205_UN,
DTL.SVC_DT              AS L2400_DTP_SVC_D8,
DTL.CLAIM_DTL_ID:::VARCHAR AS L2400_REF_PRV_CTL_NR,

1                        AS NEWROW

```

FROM SAMPL_HEADER SH

```

INNER JOIN
(
    SELECT  V1.MEMBER_ID          AS SUB_MEMBER_ID,
           V1.LAST_NM             AS SUB_LAST_NM,
           V1.FIRST_NM           AS SUB_FIRST_NM,
           V1.MEM_ID              AS SUB_ID,
           NULL                   AS SUB_DOB,
           NULL                   AS SUB_GENDER,
           NULL                   AS SUB_SSN,
           NULL                   AS SUB_ADDR,
           NULL                   AS SUB_CITY,
           NULL                   AS SUB_STATE,
           NULL                   AS SUB_ZIP,
           V1.PAYER_PROVIDER_ID   AS SUB_PAYER_PROVIDER_ID,
           V1.BILLER_PROVIDER_ID  AS SUB_BILLER_PROVIDER_ID,
           V2.MEMBER_ID          AS PAT_MEMBER_ID,
           V2.LAST_NM             AS PAT_LAST_NM,
           V2.FIRST_NM           AS PAT_FIRST_NM,
           V2.MEM_ID              AS PAT_ID,
           V2.DOB                 AS PAT_DOB,
           V2.GENDER              AS PAT_GENDER,
           V2.SSN                 AS PAT_SSN,
           V1.RES_ADDR            AS PAT_ADDR,
           V1.RES_CITY            AS PAT_CITY,
           V1.RES_STATE           AS PAT_STATE,
           V1.RES_ZIP             AS PAT_ZIP

    FROM    SAMPL_MEMBER V1
    LEFT JOIN SAMPL_MEMBER V2
    ON      V2.PAR_MEMBER_ID = V1.MEMBER_ID
    WHERE   V1.PAR_MEMBER_ID IS NULL
    AND     V2.PAR_MEMBER_ID IS NOT NULL

    UNION

    SELECT  V1.MEMBER_ID          AS SUB_MEMBER_ID,
           V1.LAST_NM             AS SUB_LAST_NM,
           V1.FIRST_NM           AS SUB_FIRST_NM,
           V1.MEM_ID              AS SUB_ID,
           V1.DOB                 AS SUB_DOB,
           V1.GENDER              AS SUB_GENDER,
           V1.SSN                 AS SUB_SSN,
           V1.RES_ADDR            AS SUB_ADDR,
           V1.RES_CITY            AS SUB_CITY,
           V1.RES_STATE           AS SUB_STATE,
           V1.RES_ZIP             AS SUB_ZIP,
           V1.PAYER_PROVIDER_ID   AS SUB_PAYER_PROVIDER_ID,
           V1.BILLER_PROVIDER_ID  AS SUB_BILLER_PROVIDER_ID,
           NULL                   AS PAT_MEMBER_ID,
           NULL                   AS PAT_LAST_NM,
           NULL                   AS PAT_FIRST_NM,
           NULL                   AS PAT_ID,
           NULL                   AS PAT_DOB,
           NULL                   AS PAT_GENDER,
           NULL                   AS PAT_SSN,
           NULL                   AS PAT_ADDR,
           NULL                   AS PAT_CITY,
           NULL                   AS PAT_STATE,
           NULL                   AS PAT_ZIP

    FROM    SAMPL_MEMBER V1
    WHERE   V1.PAR_MEMBER_ID IS NULL

) MEM
ON      1=1

INNER JOIN SAMPL_PROVIDER P1
ON      P1.PROVIDER_ID = MEM.SUB_PAYER_PROVIDER_ID

```

```

INNER JOIN SAMPL_PROVIDER P2
ON      P2.PROVIDER_ID = MEM.SUB_BILLER_PROVIDER_ID

INNER JOIN SAMPL_CLAIM CLM
ON      CLM.MEMBER_ID = COALESCE(MEM.PAT_MEMBER_ID, MEM.SUB_MEMBER_ID)

INNER JOIN SAMPL_PROFESSIONAL PRF
ON      PRF.PROF_ID = CLM.PROF_ID

INNER JOIN SAMPL_CLAIM_DTL DTL
ON      DTL.CLAIM_ID = CLM.CLAIM_ID

INNER JOIN
(
  SELECT  D.CLAIM_ID,
          SUM(CHG_AMT)           AS CLM_CHG_AMT,
          MIN(SVC_DT)           AS CLM_BEGIN,
          MAX(SVC_DT)           AS CLM_END,
          PAYER_PROVIDER_ID
  FROM    SAMPL_CLAIM_DTL D
  INNER JOIN SAMPL_CLAIM C
  ON      D.CLAIM_ID = C.CLAIM_ID
  INNER JOIN VW_SAMPL_MEMBER M
  ON      C.MEMBER_ID = M.MEMBER_ID
  GROUP BY D.CLAIM_ID,
          M.PAYER_PROVIDER_ID
) CLM_NFO
ON      CLM_NFO.CLAIM_ID = CLM.CLAIM_ID

WHERE   HDR_NAME = 'XPT_837I'

ORDER BY P2.PROVIDER_ID,
         MEM.SUB_ID,
         MEM.PAT_ID,
         CLM.CLAIM_ID,
         DTL.CLAIM_DTL_ID
);

RETURN TABLE(result);
END;
';

```

```

CREATE OR REPLACE PROCEDURE MAIN.USP_837P_EXTRACT()
RETURNS TABLE ()
LANGUAGE SQL
EXECUTE AS OWNER
AS '
BEGIN

```

```

-----
-- serenediCore - (C) 2026 Chiapas EDI Technologies, Inc.
-----

```

```

-- Increment Interchange Control Number
UPDATE SAMPL_HEADER
SET ISA_ISA13_ICN = ISA_ISA13_ICN + 1
WHERE HDR_NAME = 'XPT_837P';

```

```

-- Extraction
LET result RESULTSET := (
  SELECT  ISA_ISA02_NO_AUTH_NFO      AS X1_ISA_ISA02_NO_AUTH_NFO,
          ISA_ISA04_PSSWD,
          ISA_ISA06_MUTLY_DEF,
          ISA_ISA08_MUTLY_DEF,
          ISA_ISA11_REPTN_SEP,
          ISA_ISA12_ICN_VERS_NR,

```

```

ISA_ISA13_ICN,
ISA_ISA15_ICN_USG_IND,
ISA_ISA16_COMP_ELE_SEP,
ISA_ISA13_ICN          AS ISA_IEA02_ICN,

GSHDR_GS02_APP_SNDR_CD,
GSHDR_GS03_APP_RCV_CD,
GSHDR_GS06_GCN,
GSHDR_GS06_GCN          AS GSHDR_GE02_GCN,

(100000000 + MEM.SUB_BILLER_PROVIDER_ID)::VARCHAR
AS STHDR_ST02_TCN,
''0019''
AS STHDR_BHT01_HIER_STRUC_CD,
''00''
AS STHDR_BHT02_TS_PURP_CD,
''BAT1000000''
AS STHDR_BHT03_ORIG_APP_TRANS_ID,
CURRENT_TIMESTAMP()::TIMESTAMP_NTZ
AS STHDR_BHT04_TS_CRTN_D8,
TO_TIMESTAMP_NTZ(''1800-01-01'' || TO_CHAR(CURRENT_TIMESTAMP()::TIMESTAMP_NTZ,
''HH24:MI:SS''))
AS STHDR_BHT05_TS_CRTN_TM,
''CH''
AS STHDR_BHT06_CLM_ENC_ID,
(100000000 + MEM.SUB_BILLER_PROVIDER_ID)::VARCHAR
AS STHDR_SE02_TCN,

P2.PROV_ORG_NM          AS L1000A_NM103_NONPSNENT_NM,
P2.TAX_ID               AS L1000A_NM109_ETN_NR,
''SAMPLE''
AS L1000A_PER02_SBM_CON_NM,
''4155551212''
AS L1000A_PER04_PHN_NR,

P1.PROV_ORG_NM          AS L1000B_NM103_NONPSNENT_NM,
P1.TAX_ID               AS L1000B_NM109_ETN_NR,

P2.PROV_ORG_NM          AS L2010AA_NM103_NONPSNENT_NM,
P2.PROV_NPI             AS L2010AA_NM109_NPI,
P2.BIZ_ADDR             AS L2010AA_N301_BILL_PROV_ADDR,
P2.BIZ_CITY             AS L2010AA_N401_BILL_PVR_CITY,
P2.BIZ_STATE           AS L2010AA_N402_BILL_PVR_STAT,
(P2.BIZ_ZIP || ''0001'')
AS L2010AA_N403_BILL_PVR_ZIP,
P2.TAX_ID               AS L2010AA_REF_EMPLR_ID_NR,

''P''
AS L2000B_SBR01_PYR_RESP_SEQ_NR,
CASE WHEN MEM.PAT_MEMBER_ID IS NULL THEN ''18'' ELSE NULL END
AS L2000B_SBR02_IND_RELAT_CD,
''GRP0001''
AS L2000B_SBR04_SBR_GRP_NM,
''HM''
AS L2000B_SBR09_CLM_FIL_IND_CD,

MEM.SUB_LAST_NM         AS L2010BA_NM103_PERSN_LNM,
MEM.SUB_FIRST_NM       AS L2010BA_NM104_SBR_FNM,
MEM.SUB_ID              AS L2010BA_NM109_MEM_ID_NR,
MEM.SUB_ADDR            AS L2010BA_N301_SBR_ADDR,
MEM.SUB_CITY            AS L2010BA_N401_SBR_CITY,
MEM.SUB_STATE           AS L2010BA_N402_SBR_STAT,
(MEM.SUB_ZIP || ''0001'')
AS L2010BA_N403_SBR_ZIP,
MEM.SUB_DOB             AS L2010BA_DMG02_D8,
MEM.SUB_GENDER          AS L2010BA_DMG03_SUB_GENDR_CD,

P1.PROV_ORG_NM          AS L2010BB_NM103_NONPSNENT_NM,
P1.TAX_ID               AS L2010BB_NM109_PAYR_ID,
P1.BIZ_CITY             AS L2010BB_N401_PYR_CITY_NM,
P1.BIZ_STATE           AS L2010BB_N402_PYR_STAT,
(P1.BIZ_ZIP || ''0001'')
AS L2010BB_N403_PYR_ZIP,

CASE WHEN MEM.PAT_LAST_NM IS NOT NULL THEN ''19'' ELSE NULL END
AS L2000C_PAT01_IND_RELAT_CD,

MEM.PAT_LAST_NM        AS L2010CA_NM103_PERSN_LNM,

```

MEM.PAT_FIRST_NM	AS L2010CA_NM104_PT_FNM,
MEM.PAT_ADDR	AS L2010CA_N301_PT_ADDR,
MEM.PAT_CITY	AS L2010CA_N401_PT_CITY,
MEM.PAT_STATE	AS L2010CA_N402_PT_STAT,
MEM.PAT_ZIP	AS L2010CA_N403_PT_ZIP,
MEM.PAT_DOB	AS L2010CA_DMG02_D8,
MEM.PAT_GENDER	AS L2010CA_DMG03_PAT_GNDR_CD,
CLM.PT_CTL_NR	AS L2300_CLM01_PT_CTL_NR,
CLM_CHG_AMT::FLOAT	AS L2300_CLM02_TOT_CLM_CHG_AMT,
CLM.POS_CD	AS L2300_CLM0501_POS_CD,
'1'	AS L2300_CLM0503_CLM_FREQ_CD,
'Y'	AS L2300_CLM06_PVD_SUPP_SIG_IND,
'A'	AS L2300_CLM07_PLAN_PART_CD,
'Y'	AS L2300_CLM08_BEN_ASGT_CRT_IND,
'Y'	AS L2300_CLM09_RELS_NFO_CD,
CLM.CLM_NR	AS L2300_REF_CLM_NR,
CLM.PRIN_DIAG	AS L2300_HI0102_ICD10_PRIN_DIAG,
CLM.DIAG02	AS L2300_HI0202_ICD10_DIAG,
CLM.DIAG03	AS L2300_HI0302_ICD10_DIAG,
PRF.PROF_LNAME	AS L2310B_NM103_PERSN_LNM,
PRF.PROF_FNAME	AS L2310B_NM104_REND_PVR_FNM,
PRF.PROF_NPI	AS L2310B_NM109_NPI,
DTL.LINE_SEQ	AS L2400_LX01_ASSGD_NR,
DTL.HCPCS_CD	AS L2400_SV10102_HCPCS_CD,
DTL.DESCR	AS L2400_SV10107_DESCR,
DTL.CHG_AMT::FLOAT	AS L2400_SV102_LIN_ITM_CHG_AMT,
DTL.UNITS::FLOAT	AS L2400_SV104_UN,
DTL.DIAG_CD_PTR::INT	AS L2400_SV10701_DIAG_CD_PTR,
DTL.SVC_DT	AS L2400_DTP_SVC_D8,
1	AS NEWROW

FROM SAMPL_HEADER SH

INNER JOIN

```
(
    SELECT  V1.MEMBER_ID           AS SUB_MEMBER_ID,
           V1.LAST_NM             AS SUB_LAST_NM,
           V1.FIRST_NM           AS SUB_FIRST_NM,
           V1.MEM_ID             AS SUB_ID,
           NULL                   AS SUB_DOB,
           NULL                   AS SUB_GENDER,
           NULL                   AS SUB_SSN,
           NULL                   AS SUB_ADDR,
           NULL                   AS SUB_CITY,
           NULL                   AS SUB_STATE,
           NULL                   AS SUB_ZIP,
           V1.PAYER_PROVIDER_ID   AS SUB_PAYER_PROVIDER_ID,
           V1.BILLER_PROVIDER_ID  AS SUB_BILLER_PROVIDER_ID,
           V2.MEMBER_ID           AS PAT_MEMBER_ID,
           V2.LAST_NM             AS PAT_LAST_NM,
           V2.FIRST_NM           AS PAT_FIRST_NM,
           V2.MEM_ID             AS PAT_ID,
           V2.DOB                 AS PAT_DOB,
           V2.GENDER              AS PAT_GENDER,
           V2.SSN                 AS PAT_SSN,
           V1.RES_ADDR            AS PAT_ADDR,
           V1.RES_CITY            AS PAT_CITY,
           V1.RES_STATE           AS PAT_STATE,
           V1.RES_ZIP             AS PAT_ZIP

    FROM    SAMPL_MEMBER V1
    LEFT JOIN SAMPL_MEMBER V2
    ON      V2.PAR_MEMBER_ID = V1.MEMBER_ID
```

```

WHERE V1.PAR_MEMBER_ID IS NULL
AND V2.PAR_MEMBER_ID IS NOT NULL

```

```

UNION

```

```

SELECT V1.MEMBER_ID           AS SUB_MEMBER_ID,
       V1.LAST_NM             AS SUB_LAST_NM,
       V1.FIRST_NM           AS SUB_FIRST_NM,
       V1.MEM_ID             AS SUB_ID,
       V1.DOB                AS SUB_DOB,
       V1.GENDER             AS SUB_GENDER,
       V1.SSN                AS SUB_SSN,
       V1.RES_ADDR           AS SUB_ADDR,
       V1.RES_CITY           AS SUB_CITY,
       V1.RES_STATE          AS SUB_STATE,
       V1.RES_ZIP            AS SUB_ZIP,
       V1.PAYER_PROVIDER_ID  AS SUB_PAYER_PROVIDER_ID,
       V1.BILLER_PROVIDER_ID AS SUB_BILLER_PROVIDER_ID,
       NULL                  AS PAT_MEMBER_ID,
       NULL                  AS PAT_LAST_NM,
       NULL                  AS PAT_FIRST_NM,
       NULL                  AS PAT_ID,
       NULL                  AS PAT_DOB,
       NULL                  AS PAT_GENDER,
       NULL                  AS PAT_SSN,
       NULL                  AS PAT_ADDR,
       NULL                  AS PAT_CITY,
       NULL                  AS PAT_STATE,
       NULL                  AS PAT_ZIP
FROM   SAMPL_MEMBER V1
WHERE  V1.PAR_MEMBER_ID IS NULL

```

```

) MEM
ON 1=1

```

```

INNER JOIN SAMPL_PROVIDER P1
ON P1.PROVIDER_ID = MEM.SUB_PAYER_PROVIDER_ID

```

```

INNER JOIN SAMPL_PROVIDER P2
ON P2.PROVIDER_ID = MEM.SUB_BILLER_PROVIDER_ID

```

```

INNER JOIN SAMPL_CLAIM CLM
ON CLM.MEMBER_ID = COALESCE(MEM.PAT_MEMBER_ID, MEM.SUB_MEMBER_ID)

```

```

INNER JOIN SAMPL_PROFESSIONAL PRF
ON PRF.PROF_ID = CLM.PROF_ID

```

```

INNER JOIN SAMPL_CLAIM_DTL DTL
ON DTL.CLAIM_ID = CLM.CLAIM_ID

```

```

INNER JOIN
(

```

```

    SELECT D.CLAIM_ID,
           SUM(PMT_AMT)           AS CLM_PMT_AMT,
           SUM(CHG_AMT)           AS CLM_CHG_AMT,
           SUM(COALESCE(COPAY, 0.00) + COALESCE(COINS, 0.00) + COALESCE(DEDUCTIBLE, 0.00)) AS
CLM_PR_AMT,
           MIN(SVC_DT)           AS CLM_BEGIN,
           MAX(SVC_DT)           AS CLM_END,
           PAYER_PROVIDER_ID
FROM   SAMPL_CLAIM_DTL D
INNER JOIN SAMPL_CLAIM C
ON D.CLAIM_ID = C.CLAIM_ID
INNER JOIN VW_SAMPL_MEMBER M
ON C.MEMBER_ID = M.MEMBER_ID
GROUP BY D.CLAIM_ID,
         M.PAYER_PROVIDER_ID

```

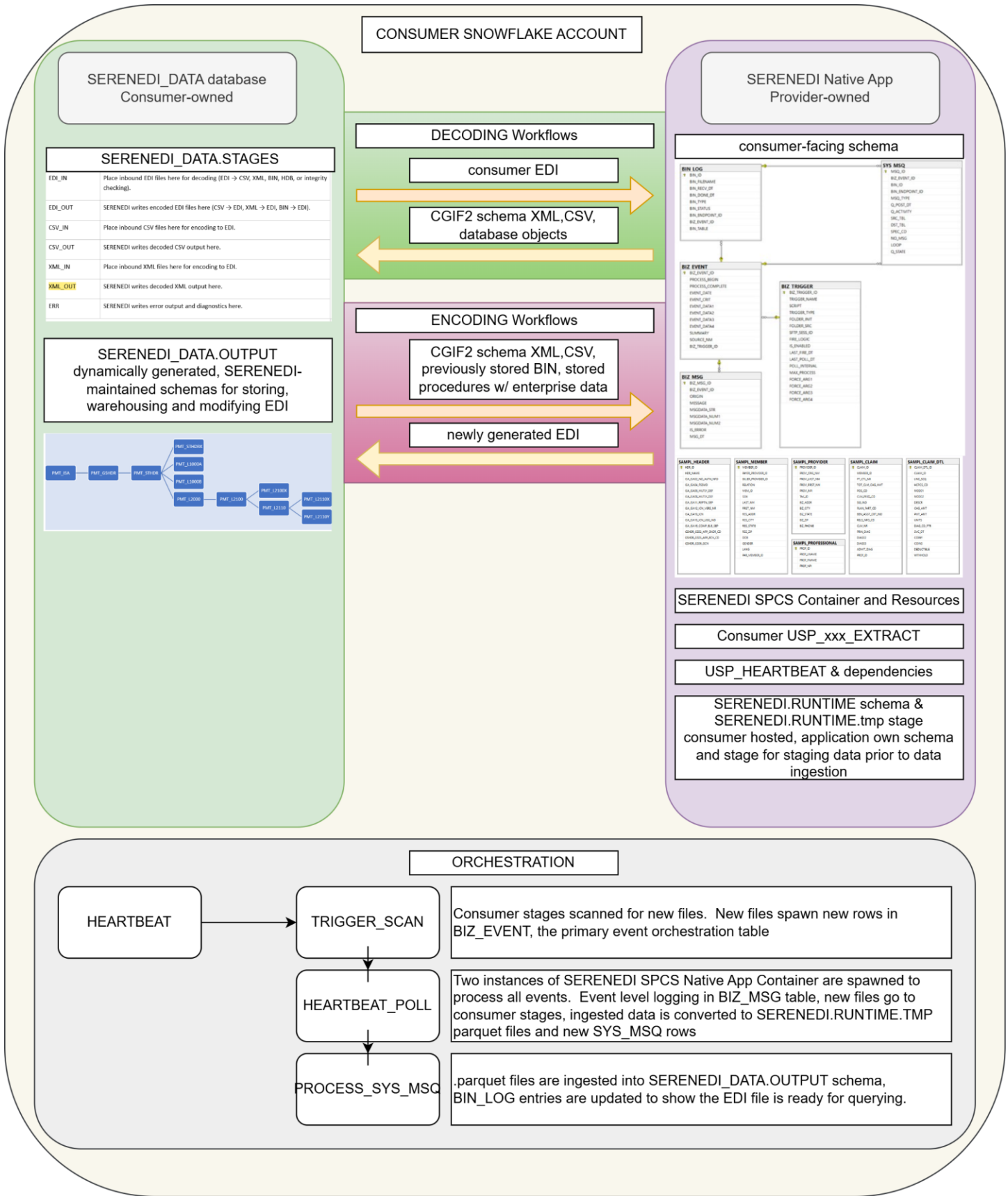
```
) CLM_NFO
ON      CLM_NFO.CLAIM_ID = CLM.CLAIM_ID

WHERE   HDR_NAME = ''XPT_837P''

ORDER BY P2.PROVIDER_ID,
         MEM.SUB_ID,
         MEM.PAT_ID,
         CLM.CLAIM_ID,
         DTL.CLAIM_DTL_ID
);

RETURN TABLE(result);
END;
';
```

SERENEDI Architecture



This diagram shows the core tables for the SERENEDI Snowflake Application. 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.

Data Ingestion

Whenever SERENEDI processes events linked to the BIN or HDB triggers, whether arriving via the two workflows or with manually inserted BIZ_EVENT records, it goes through the following process.

For *FLAT* data ingestions:

1. A new BIN ID is generated that uniquely identifies the data stored with this command. A single file is associated with a single BIN_ID. For Flat destinations, a BIN_ID is associated with every row in the destination table. For HDB destinations, the BIN_ID is inserted for every loop belonging to that file.
2. A set of new records is created in the serenediCore SYS_MSQ table as follows:
 - a. Flat Merge operations: Q_ACTIVITY = 'FLAT_MERGE', Q_STATE = 'U'
 - b. Flat ForceMerge operations: Q_ACTIVITY = 'FLAT_FORCEMERGE', Q_STATE = 'U'
 - c. MSQ_TYPE = 'DATA_SHUTTLE'
 - d. SRC_TBL receives the full path to the generated .parquet file
 - e. DST_TBL receives the *destination table* name
3. The data is stored on a .parquet file generated in SERENEDI.RUNTIME.TMP/{hash}_{loop name}/{event ID}_{table prefix}_{binID}_{random GUID}.parquet. The hash is based off the destination table and ensures that .parquet files associated with a single destination table are present within the same folder.

For *HDB* data ingestions:

1. A new BIN ID is generated by the distribution database that uniquely identifies the data stored with this command.
2. Parquet files are generated in SERENEDI.RUNTIME.TMP/{shortHash}_{short loop name}/{event ID}_{prefix}_{short loop name}_{bin ID}_{GUID}.parquet
3. For each parquet file generated, there will be one SYS_MSQ entry:
 - a. HKey ForceMerge operations: Q_ACTIVITY: 'FLAT_FORCEMERGE', Q_STATE='U'
 - b. MSQ_TYPE = 'DATA_SHUTTLE'
 - c. SRC_TBL receives the full path to the .parquet file associated with this file and this loop
 - d. DST_TBL receives the HDB destination table name
 - e. The SPEC_CD and LOOPNM are set to match the destination specification and the name of the loop (sans the 3 digit specification prefix)

PROCESS_SYS_MSQ is called by the main Heartbeat stored procedure once all of the events have been processed; this is important to allow all of the .parquet files to be generated prior to processing. Once it does launch, it scans the SYS_MSQ and processes the entries grouped by the destination table. It makes two calls: PROCESS_DST_TBL and PROCESS_DST_INSERT, which get passed a single value: the current destination table being processed.

1. In PROCESS_DST_TBL, it checks if the destination table does not exist:
 - a. Scan the first .parquet file and create the table from that file's specification
 - b. Create a primary key constraint named PK_{table name} bound to the two columns (BIN_ID and BIN_IX)
 - c. Ensure the table is accessible to the APP_PUBLIC role
 - d. For HDB inserts that are *not* the ISA loop, it will

- i. Create an unenforced constraint FK_{table name} on the (BIN_ID, PAR_BIN_IX) tied to the *parent* table of this loop. This is a “hint” to schema scanning Snowflake programs to indicate the parent/key relationship in the schema.
 - ii. Log a BIZ_MSG entry where BIZ_EVENT_ID =0, MSG_ORIGIN = “PROCESS_DST: CREATED TABLE”, and MESSAGE = parent table.
 - iii. If this is a 2300 loop belonging to the 837 D / I / P specifications, one additional foreign key is created: FK_{table name}_2000C linking the fields (BIN_ID, PAR_2000C_IX) to the (BIN_ID, BIN_IX) fields on a 2000C Patient Data loop. This is another “hint” constraint that shows that the 2300 Claim loop has relationships to both its parent subscriber 2000B loop *and* an optional patient 2000C loop. When the PAR_2000C_IX loop is null, the claim is a subscriber claim. When the PAR_2000C_IX loop joins to a BIN_IX of the L2000C parent table, then the claim joins to both the subscriber (BIN_IX) *and* the patient (PAR_2000C_IX) indicated in the patient loop.
2. In PROCESS_DST_INSERT, the following steps are executed on each destination table:
 - a. First, the stored procedure executes a COPY INTO statement which loads the .parquet files into the destination table. As they are processed, they are deleted.
 - b. Next, all of the SYS_MSQ entries associated with the processed .parquet files are set to a Q_STATE of ‘C’ (“complete”) and Q_POST_DT is set to the current timestamp.
 - c. If there’s a failure for any reason, that SYS_MSQ entry is marked as a “Z”.
 - d. Any failures will be logged to the BIZ_MSG table with a BIZ_EVENT_ID of 0, and a MSG_ORIGIN of “PROCESS_DST_INSERT:FILE_ERROR”, and MESSAGE indicating the file(s) that failed to process.
3. Next, UPDATE_BIN_STATUS() is called. This will do the following:
 - a. If there are any SYS_MSQ.Q_STATE = ‘U’, it will set the matching BIN_ID to a status of PENDING.
 - b. If there are any SYS_MSQ.Q_STATE = ‘C’ *and* the number of completed entries matches the original record count inserted for the BIN, then the BIN_STATUS is set to COMPLETE.
 - c. If there are any SYS_MSQ.Q_STATE = ‘Z’, that BIN_LOG is set to ERROR.
 - d. For all errored BIN_IDs, any *partially inserted* BIN or HDB records will be culled from the destination tables.
 - e. All completed SYS_MSQ entries are purged from the SYS_MSQ table.

System Configuration

The SERENEDI Snowflake environment is configured simply through global values set in a single SERENEDI.MAIN.APP_CONFIG table. There are four global values:

CONSUMER_OUTPUT_PREFIX – This is the target schema used for housing database objects generated by SERENEDI BIN and HDB decode operations. This is configured during the consumer-side initialization script with a default set to SERENEDI_DATA.OUTPUT.

CONSUMER_STAGES_PREFIX – This is the schema used to hold SERENEDI’s workflow stages. By default, it is set to SERENEDI_DATA.STAGES.

CONSUMER_ROLE – The consumer role used to assign the SERENEDI permissions. By default, it is set to the CURRENT_ROLE() function that matches the role when SERENEDI was installed.

CONSUMER_SEGPOOL_FORMAT – This is an optional variable controls a number of global flags that control various SERENEDI behaviors. If it is not set, the default value is “*~: ^YYSNIP_CODES”

These correspond to the follow global values:

Position	Function
1	EDI Encoding - Element terminator
2	EDI Encoding – Segment terminator
3	EDI Encoding – Composite Element terminator
4	EDI Encoding – Repeating Element terminator
5	EDI Encoding – Y/N – Terminate segments with a Carriage Return
6	EDI Encoding – Y/N – Terminate segments with a Line Feed
7+	EDI Decoding – These values set the behavior of SERENEDI’s EDI decoder and what kind of messages it logs to the BIZ_MSG table during operation: NOMSG – Normal SNIP 1/2 integrity messages will be omitted, only critical errors will be logged CODES – CODESET validation messages plus SNIP 1/2 integrity messages SNIP_CODES - – Full SNIP validation messages, no codeset checks (1-2 for all specifications, and Types 1-5 for those specifications that support further validations – 270,271,834,835,837I and 837P) SNIP – Full SNIP validation messages, no codeset checks (1-2 for all specifications, and Types 1-5 for those specifications that support further validations – 270,271,834,835,837I and 837P)

Integrity Validations

Code Set Checks

Code Set	Source
Claim Adjustment Reason Codes	Washington Publishing Company
Remittance Adjust Reason Codes	Washington Publishing Company
Claim Frequency Codes	Washington Publishing Company

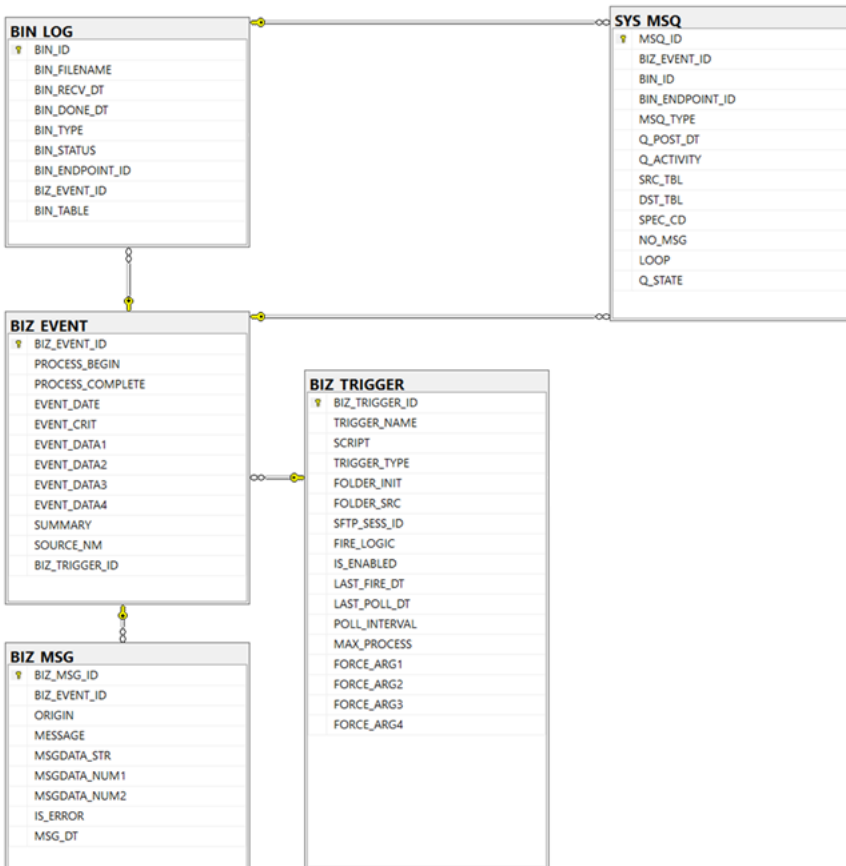
ICD-9-CM Diagnosis Codes	CMS
ICD-9-PCS Procedure Codes	CMS
ICD-10-CM Diagnosis Codes	CMS
ICD-10-PCS Procedure Codes	CMS
National Drug Codes	CMS
Provider Taxonomy Codes	Washington Publishing Company
State Abbreviation Codes	United States Post Office
ISO Country Codes	United States Post Office

These code sets are updated on a twice-a-year basis on or near the first of February and September. These updates to the SERENEDI Application will occur in the background and should normally be transparent to consumers.

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



SERENEDI runs SNIP Type 1 and 2 validations automatically across all file decodes. 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, and only for the supported specifications of 270, 271, 834, 835, 837P, 837I.

SERENEDI Core Schema

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_ID, often referred to as the BIN ID in this manual. The flat and hierarchical data stores are with SQL queries. Until the BIN_STATUS is set to COMPLETE, the data will not be ready to query, and the data is still getting loaded. If the BIN_STATUS is set to ERROR, that means the data will not be loaded at all.

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 <i>prefi102x</i> 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_ENDPOINT_ID	Integer (FK)	<i>Unused</i>
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.

BIZ_TRIGGER Table

This is a core table that defines the source folders used in the different workflows. Note that as this schema is inherited from another version of the technology, most of the columns are unused.

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)	<i>Unused</i>

TRIGGER_TYPE	Varchar(20)	This is defined as follows: PASSIVE: The trigger will not actively fire events, but is used to link user-generated events to a workflow. 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.
FOLDER_INIT	Varchar(200)	For LOCAL_UPLOAD triggers, users drop the file in the folder indicated in FOLDER_INIT.
FOLDER_SRC	Varchar(200)	This is used by the LOCAL_UPLOAD triggers as the destination to move files to prior to firing the trigger.
SFTP_SESS_ID	Integer (FK)	<i>Unused</i>
FIRE_LOGIC	Varchar(4000)	<i>Unused</i>
IS_ENABLED	Integer	When this value is 1, the trigger is enabled. Any other value will disable the trigger.
LAST_FIRE_DT	DateTime	<i>Unused</i>
LAST_POLL_DT	DateTime	<i>Unused</i>
POLL_INTERVAL	Integer (not null)	<i>Unused</i>
MAX_PROCESS	Integer	<i>Unused</i>
FORCE_ARG1	Varchar(200)	<i>Unused</i>
FORCE_ARG2	Varchar(200)	
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 SPCS Container workers. Most of the time, this unit of work is centered around a single file moving through the EDI pipeline via the Upload triggers, but it can also drive work through user-generated events inserted into this table.

Events are “owned” by worker processes by updating the SUMMARY with a unique ID. Once the event has been taken over by a container instance, the summary is set to “STARTING”. Events are not processed in sequential order but rather in a random order.

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.

Field Name	Data Type	Purpose
BIZ_EVENT_ID	Integer (PK)	This is 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 <i>criteria</i> , generally the filename or some other string that fired the triggering mechanism.
EVENT_DATA1	Varchar(1000)	These event <i>data</i> fields provide data about the feeds to the SCORE script linked to the trigger that created this event. The usage of these fields is dependent on the specific workflow associated with the event.
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.
SOURCE_NM	Varchar(1000)	This indicates the type of trigger that created the event.
BIZ_TRIGGER_ID	Integer	This is a foreign key reference to an item in the BIZ_TRIGGER table.

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 unenforced foreign key to the BIZ_EVENT_ID that spawned this message. If it is 0, then this will be an error generated by the BIN system and not tied to a specific event.
ORIGIN	Varchar(50)	<p>This is a short string that denotes the origin of the message. Possible values are:</p> <p>DATA_SHUTTLE – This denotes fields that were added by the BIN system or flagged as not present.</p> <p>USER – This indicates a message generated by a SCORE script and given the default ORIGIN.</p> <p>For the syntax error messages, each “family” of errors will have a different Origin.</p>

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.

SYS_MSQ

The function of the SYS_MSQ table is explained in the earlier 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.

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.
BIN_ID	Integer (FK)	This is a foreign key to the BIN_LOG table, and specifies the BIN_ID this operation is inserting to the destination table.
BIN_ENDPOINT_ID	Integer	<i>Unused</i>
MSQ_TYPE	Varchar(20)	This value is hard-coded as DATA_SHUTTLE.
Q_POST_DT	DateTime	If there is a critical error, this is the time the record errored out. This is the only time that SYS_MSQ records are persisted for any length of time.
Q_ACTIVITY	Varchar(50)	This value is hardcoded to be: FORCE_FLATMERGE
SRC_TBL	Varchar(200)	This specifies the .parquet file source of the shuttle request
DST_TBL	Varchar(200)	This specifies the <i>destination table</i> of the shuttle request.
SPEC_CD	Varchar(2)	This is the two digit specification code of the specification used for this merge/ force merge operation.
NO_MSG	Varchar(1)	This is a flag to indicate if messages are to be suppressed.
LOOP	Varchar(30)	This is used only for hierarchical operations, and indicates the loop short name used for the merge / force merge operation.

Q_STATE	Varchar(1)	This is a single-character status code: U – HDB/Flat data table is ready for transfer to the destination Z – A critical error occurred while processing this request
----------------	-------------------	--

SERENEDI Incident Response Plan

Chiapas EDI Technologies, Inc. • Davis, California

1. Purpose

This document defines the incident response process for security vulnerabilities and operational incidents affecting the SERENEDI Snowflake Native App and its Snowpark Container Services (SPCS) container image. It establishes severity classifications, response timelines, communication procedures, and remediation commitments.

2. Scope

This plan covers the following components of the SERENEDI product:

- The SERENEDI SPCS container image (serenedi:latest)
- All stored procedures deployed via the Native App setup script
- The SERENEDI application package and its configuration
- Provider-side infrastructure used in building and distributing the application

Consumer-owned resources (the SERENEDI_DATA database, consumer compute pools, and consumer warehouses) are under the consumer's operational control and are outside the scope of this plan.

3. Severity Classification and Response SLAs

Severity	Definition	Initial Response	Remediation Target
Critical / High	Active exploitation, data exposure, unauthorized access to consumer data, or exploitable CVE in container image or application code	24 hours	72 hours
Medium	Vulnerability requiring specific conditions to exploit, or non-critical security hardening issue	72 hours	Next scheduled release
Low	Informational finding, best-practice deviation, or minor hardening improvement	5 business days	Next scheduled release

4. Incident Response Process

4.1 Detection and Reporting

Security incidents may be identified through:

- CVE scans of the SERENEDI container image (performed prior to each release)
- Dependency vulnerability alerts from upstream packages
- Reports from consumers or Snowflake's security team
- Snowflake's automated Native App security scanning pipeline

Consumers and other parties may report security concerns to support@chiapas-edi.org.

4.2 Assessment and Triage

Upon identification of a potential vulnerability, Chiapas EDI will assess the severity using the classifications above, determine affected components and versions, evaluate whether the SERENEDI architecture mitigates the risk (e.g., no-egress design, consumer-owned data isolation), and assign a severity level and initiate the corresponding response timeline.

4.3 Remediation

Remediation actions include:

- Patching the container base image and rebuilding
- Updating application dependencies to non-vulnerable versions
- Modifying stored procedures or application logic as required
- Publishing an updated application version through Snowflake's Native App update mechanism

4.4 Communication

For Critical and High severity incidents, affected consumers will be notified via email within the initial response window. Notifications will include a description of the vulnerability, affected versions, recommended actions, and expected remediation timeline. Security advisories will be published on chiapas-edi.org as appropriate.

4.5 Post-Incident Review

Following resolution of Critical or High severity incidents, Chiapas EDI will conduct a post-incident review to identify root cause, evaluate whether additional preventive measures are warranted, and update this plan if necessary.

5. Architecture Risk Mitigation

SERENEDI's architecture inherently mitigates several classes of security risk:

- **No egress:** The SPCS container makes no outbound network connections. There are no external access integrations, egress rules, or network rules.
- **Consumer data isolation:** All consumer data remains within the consumer's Snowflake account. SERENEDI does not transmit, copy, or store consumer data outside the consumer account.

- **No external dependencies at runtime:** The container does not download code, models, or configuration at runtime. All components are bundled in the application package.
- **Snowflake-native authentication:** SERENEDI uses Snowflake's built-in OAuth token for SPCS container authentication. No custom authentication or credential storage is implemented.

6. Contact Information

Security and Support Contact: support@chiapas-edi.org

Organization: Chiapas EDI Technologies, Inc.

Location: Davis, California

Website: <https://chiapas-edi.org>