chiapasCore

Licensing

Chiapas V1.0 - HIPAA Translation System

Chiapas was built upon the Microsoft .NET framework.
Microsoft .NET Framework 1.1

Chiapas utilizes SQLite technology provided by Dr. D. Richard Hipp, released into the public domain with the following notice:

The author disclaims copyright to this source code. In place of a legal notice, here is a blessing:

May you do good and not evil.
May you find forgiveness for yourself and forgive others.
May you share freely, never taking more than you give.

Chiapas uses SQLite C# interface technology from Finisar Corporation.
The Finisar license follows:

Chiapas uses the AWK text parser program owned by Lucent Technologies.

Table of Contents

Chapters

Appendixes

Chapter 1: Introduction

This manual relates to the process of using the Chiapas technology to interact with modern healthcare business systems. Specifically, it translates business data to and from the HIPAA protocol. Due to the complexity of this protocol, there is a steep learning curve before Chiapas can be used with business data. Therefore, the audience for which this manual was written is primarily healthcare IT professionals with knowledge of both general healthcare IT functioning and some general knowledge of the HIPAA protocol itself.

In this introduction, I will outline the vision driving the development of this software. Then, I will explain some background of the Healthcare Insurance Portability Administration Act, give a brief overview of healthcare EDI, and finally cover the Implementation Guides and how Chiapas relates to them.

Chiapas

Chiapas was designed to empower healthcare organizations by giving them a software engine focused on transporting data between business data and the HIPAA protocol. Just as there is a diversity of healthcare implementations, Chiapas provides a diverse selection of interfaces to interact with business data. Furthermore, Chiapas provides a thorough HIPAA mapping system suitable to map data to and from HIPAA specifications. Finally, Chiapas provides an embedded SQL engine for common healthcare data shaping actions.

History of HIPAA

The Healthcare Insurance Portability Administration Act was passed by Congress on August 21, 1996 and signed into law by President George W. Bush on December 27, 2001. This law was introduced to compel U.S. healthcare organizations to adopt a standardized method of electronic data interchange (EDI), as well as to protect the privacy and security of member patient data. At the time the bill was introduced, hundreds of file formats existed between provider organizations to exchange business data.

Although a number of mandates relating to security and privacy were introduced in HIPAA, Chiapas is focused only on the "administrative simplification" provisions, specifically, on the HIPAA Implementation Guides (HIGs) and integrating them into healthcare business systems. The HIGs consist of a number of documents outlaying the exact transmission protocol used to relay information relating to membership, claims, financial remittance, and eligibility. They are available to the general public from the Washington Publishing Company at www.wpc-edi.com.

Because of the complexity of the protocol, many healthcare organizations consider it impractical to make their business systems compliant with the HIPAA protocol on their own.

Furthermore, there exists a large diversity in the systems used by clinics and HMOs around the nation. Not only this, but the intellectual capital needed to bring any single organization into compliance is in short supply. Chiapas was designed as a middleware solution to this problem. As it provides a variety of data import and export mechanisms, most providers should be able to find a "critical path" that allows their business systems to exchange meaningful business information with Chiapas and then the HIPAA protocol. Chiapas has been used to import from many types of data files, including HCFA-1450 printer dump files and many other types.

Implementation

Once a data transport mechanism between the business system and Chiapas is laid out, it is necessary to work out the interface between Chiapas and HIPAA. At this time, someone with a firm understanding of the trading partner agreements and the business logic will need to step in and design the data shaping and mapping layers used by Chiapas. The mapping layer refers to translating business fields to fields within a HIPAA file, where the data shaping layer refers to manipulating data within business fields from or to HIPAA data. Tools are provided within Chiapas to make this easier, as well as several tutorials to approach these problems step-by-step.

Critical Path

The term critical path here refers to the sequence of steps and challenges to overcome before HIPAA is successfully installed at a business site. For Chiapas, the implementor must understand the business data and business systems; he or she must know about data shaping and how it relates to HIPAA compliance. Then, a thorough knowledge of the Hierarchy Tree and how it relates to HIPAA mapping is needed. Finally, the actual mechanics of encoding and decoding need to be understood.

Chapter 1 is a broad overview of Chiapas, its components and how it would be used in a business. Chapter 2 covers the HIPAA Hierarchy Tree and how it represents the HIPAA Implementation Guides. Chapter 3 covers data shaping, which is the transformation of business data to and from a HIPAA-suitable equivalents. Chapter 4 is an overview of mapping. Chapter 5 covers the Chiapas Encoder and how mappings and the Hierarchy Tree join together to create HIPAA files. Chapter 6 covers decoding. Chapter 7 covers the Chiapas scripting language, JobScript. In the appendices are mapping tutorials relating to claims, enrollment and remittance activity; an error glossary; and the JobScript reference guide.

Launching Chiapas

Chiapas can be run three different ways. Initially, it can be launched via the Chiapas Studio application that provides a front-end to the major functional components and operations of Chiapas. This can be used to develop mapping keys and test them with business processes.

Chiapas can also be run in ‘shell’ mode. This is done by running the chiapasMain.exe with the ‘shell’ argument.

```
chiapasMain.exe shell
```

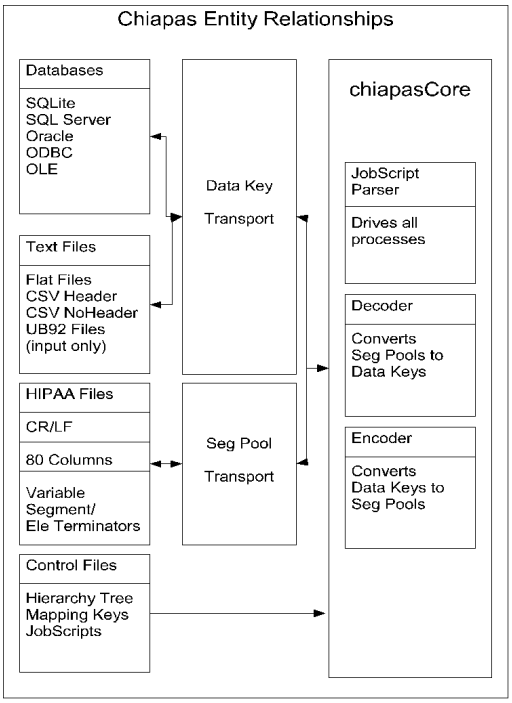
This will allow you to run scripts and enter in Script commands. It should be noted that looping, logic branching and other ‘@’ prefixed commands are disabled except for use in the PARSE command.

Finally, Chiapas can be run in ‘Clearinghouse’ mode. In this mode, you must pass the root Chiapas directory as an argument. For example,

```
ChiapasMain.exe c:\edi\version1
```

Chiapas will continue polling the SCRIPTS_PENDING subdirectories for scripts; if encountered, it will execute them and place them in the SCRIPTS_COMPLETE sub directory. Scripts inside the SCRIPTS_PERM directory will be repeatedly executed in a separate thread.

Chapter 2: Overview



Chapter Summary:
1. Define Business Requirements
2. Create an Inventory
3. Establish Links to Business System
4. Mapping Layer
5. Data Shaping Layer
6. Integrity and Verification
7. Deployment and Testing
8. Production

In this chapter, I will cover the sequence of steps an organization needs to take in order to transport their business data to and from HIPAA using Chiapas. In order to get the best use of the technology, the organization needs a clear business vision of the critical path. Here, I am presenting a critical path of eight discrete steps.

1. Define Business Requirements

Who are the trading partners? What are their requirements and what are yours?

The first steps to a successful HIPAA deployment for a business is to define what needs to be done and who needs to do it. All of the trading partners need to be identified, and then an open discussion needs to occur with them to work out a time line for implementation.

Specific goals need to be worked out and definite target dates established for all of the various stages. For new Chiapas users, it is helpful if the actual parties responsible for implementation use the software with the supplied sample data files and go through the tutorials. This will give them a much stronger idea of how long it will take to achieve things with Chiapas.

2. Create an Inventory

What data is present in the business system? In what formats can this data be exported? How can this data be imported? Does the data need extensive data scrubbing before being imported or exported?

When exporting business data to HIPAA for a trading partner, it must first be assessed exactly how the data will be extracted from the sender's business system on a day to day basis. Then, the business data needs to be analyzed: Is it organized around a strict adherence to business IDs and high data quality, or does an extensive amount of “data scrubbing” need to be done first? The resulting HIPAA file is only going to be as good as the data source.

Usually, importing data from HIPAA is simpler. The act of exporting to a valid HIPAA file usually implies a certain baseline of data quality. After that, it's usually a matter of assessing how to

translate the incoming business data to records inside the receiver's business system.

3. Establish Links to Business System

Before the data can be processed, it must be submitted in an acceptable form. Chiapas presents a rich set of options for data transport in order to be compatible with a wide variety of healthcare business systems.

Import and Export Data formats:
- Comma Separated Values (CSV) format, with and without column name header row
- Fixed-width Flat Files
- SQL Server database tables
- Oracle database tables
- ODBC-interface database tables

Import Only Data Formats:
- HCFA-1450 Printer dump-to-file
- UB92 6.0 data format

Import and Export HIPAA formats:
- 80 column format
- Any segment and element separator characters
- Optional CR, LF or both appended to each segment

Chiapas provides a simple scripting language called “JobScript” that contains many commands relating to the import of export of data based on these formats. JobScripts may be executed in a number of ways, as covered in Chapter 8: JobScripts.

4. Mapping Layer

Mapping is the process of matching one set of data elements or individual code values to their closest equivalents in another set of them. The Mapping Layer here then is a layer of abstraction between a business system and the data contained within a HIPAA file.

Chiapas mappings represent the translation of a single business table to and from a given set of HIPAA fields. These mappings are coalesced into something called a Map Key. A single map key represents a translation lookup between a business table or file and a HIPAA file. Except for a few special cases, it is bi-directional, meaning it can be used in both directions.

The HIPAA Implementation Guides rigidly define a protocol for use in business interchanges. However, within this protocol are almost limitless variations on how it can be used for any given purpose. For example, both segments and loops can often be present for multiple iterations, so the mapping system in Chiapas must be just as flexible. It can not only focus on a particular segment within a specific loop, but also map to a specific repeat of a segment. This flexibility allows Chiapas to adapt to a wide variety of trading partner scenarios.

5. Data Shaping Layer

Mapping is challenging because you need extensive knowledge of the HIPAA Implementation Guides to know what each business field represents. However, the data as-is may not necessarily be HIPAA compliant. For example, let us say a Line Service cost $85.10, and the business system stores this as a text field of '85.10.' This is not technically HIPAA compliant, because HIPAA does not allow trailing zeros—this would need to be converted to '85.1' before it can be encoded to a valid HIPAA file. Thus, there's a need for another layer between a business system and the Mapping Layer: the data shaping layer.

Initially, the data may not be suitable for a smooth conversion to a HIPAA file. As described above in the Inventory section, business data that is “non-normalized” will confuse the Chiapas Encoder engine and split up the data in unusual ways. The process of data shaping acts as a buffer so that before the data is presented to the Encoder, it is converted to a form appropriate for encoding.

In the opposite direction, data shaping can take data extracted from HIPAA file and convert it into a form better suited for business use. In HIPAA, all diagnosis codes should not have an embedded period. Data shaping can reinsert this period so that it is again suitable for the target business system. Also, all HIPAA dates are in an eight digit YYYYMMDD (year month day) format that may not be compatible with the Date/Time fields of a target SQL database system. It is the data shaping layer that presents the HIPAA data back in a form acceptable by the target.

6. Integrity and Verification

The resulting file needs to be tested against a number of quality requirements. Chiapas can verify level 1 and 2 HIPAA integrity, but there are a number of additional requirements that make up a valid data file. The objective for Chiapas is to check a certain baseline of data integrity defined by the basic syntax of the HIPAA structure.

As part of this objective, Chiapas has the ability to do a certain amount of low-level validation of a HIPAA file. Since there is an exact specification representation given in the Hierarchy Tree, Chiapas uses the information contained therein to do a series of quality checks on the data.

These checks can ensure:

- Element data type compliance: Date fields must only contain valid dates; number fields contain only numbers; 'AN' ID fields may contain anything; etc.
- Required and Situational element enforcement
- Element value list compliance: If a list of specified values is present, than only one of those values must be present
- Required and Situational segments within a loop
- Required loops are present
-

These quality checks ensure a certain baseline of quality; it does NOT guarantee that the file is truly HIPAA compliant. Chiapas does not come with a list of valid procedure codes or ICD-9 diagnosis codes, so it is the responsibility of the user to ensure HIPAA compliant business information is being transmitted. Also, there are numerous business rules specified within the HIPAA Implementation Guides that are beyond the scope of Chiapas' purpose. For this reason, Chiapas is best used with an Internet-based HIPAA testing bureau that can give immediate feedback on a certain files' level of compliance.

7. Deployment and Testing

After mapping and data shaping rules have been established, and the files are verified to be HIPAA compliant, it is time to shift into testing and then to deploy to production. As part of this, there should be contingency plans established for critical failures in case a file cannot be decoded. There are methods that JobScript can detect critical failures, but it should be decided upon ahead of time where these are handled.

Furthermore, all incoming and outgoing data should be assessed via the business requirements, preferably with large files that can uncover any hidden weaknesses in the setup. For extra safety, Chiapas can be set up to detect any critical errors in the files it encodes before submitting them for transport.

Once a process is successfully deployed, it may work fine for a few months and then break suddenly. This is almost always because of an error in the source data, or a breakdown in the data shaping 'normalization' process.

If the Internet is used to submit the HIPAA file to the recipient via an unencrypted channel of communication (via e-mail, Web site or FTP), the data must be encrypted. A testing bureau may be required to establish the validity of the file. A testing bureau may also be used to establish a baseline of HIPAA validity.

Either direct Chiapas API calls or JobScripts can be written to meet production needs. This includes logging, activity record keeping and moving files out of “pending” bins.

Finally, it is important to note that HIPAA dictates how a file carrying confidential patient data is to be transported. If it is submitted via the Internet, it MUST be encrypted. Files submitted via dial-up phone systems are not subject to this requirement. Chiapas does not encrypt files itself—this is left to be implemented by the trading partners.

8. Production

At this time, the trading partners can go live with HIPAA submissions. Once a steady production process is established, all HIPAA file generators should establish a HIPAA Companion Guide. This lays out exactly which fields are sent in a HIPAA file, and what each element maps to on the sender's business system. This way, an authoritative source of information is created that dictates exactly how the file needs to be processed by the receiver. In times of changing personnel and data systems, this guide can be a 'Rosetta Stone' should the data systems need to be redone.

Decoding from HIPAA to a Business System

Chiapas can turn a HIPAA file into a single flat table through a mapping and decoding pass. Given a correct mapping file and a HIPAA file, Chiapas will parse the incoming information into a flat table. The number of columns will equal the number of mapping fields. The number of rows correlate to how the primary keys are set up in the mapping, but typically there will be a new row for every claim service line or similarly low-level information.

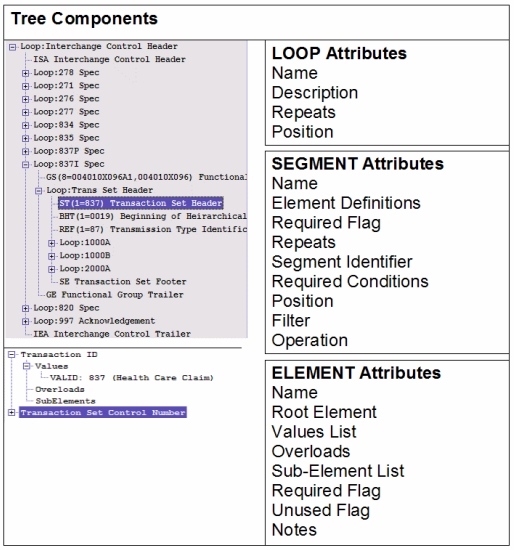
Encoding to HIPAA from a Business System

The data must be presented to Chiapas in a form described similar to above, with one important addition: definitions. Encoding requires much more information than decoding! During the decode process, you may selectively ignore many required fields, but during encoding, certain pieces of data must be defined to create a valid file. This may include contact information about the submitter, business IDs about the sender and receiver, and so on.

Another critical requirement of this information is that the data is “normalized.” For example, if your business system contains two entries with the same patient control number (aka Claim ID), and there are two separate spellings of that patient's name, Chiapas will interpret this data as two separate claims belonging to two separate people. The reasons for this will become clearer later on, but it relates to how incoming data is rearranged prior to encoding. Basically, there cannot be subtle variations in similar pieces of data within instances of a specified primary key. More on this will be explained in Chapter 6: Encoding Process.

Chapter 3: Hierarchy Tree

Chapter Summary:
- Loops are the main structure organizing segments
- Segments organize related fields of information
- Elements are the most atomic container of data



A pivotal point to understanding Chiapas and how it interacts with business systems is the Hierarchy Tree. This Tree - represented as a single file in the default Chiapas setup - contains all supported data from the HIPAA Implementation Guides. It is called a Tree because it follows a branching, leaf-like structure similar to the HIPAA format itself. The levels alternate between a "Loop" and a number of segments or sub loops. The topmost several loops in the tree are called envelopes.

At the top of the Tree is the Outer Interchange envelope, the ISA loop. Within this loop are the ISA and IEA segments, and in between them, a new sub-loop representing every supported specification. Each sub-loop represents a new specification, itself contained within a GS/GE envelope. Within that sub-loop will be further loops representing the ST/SE envelope and all associated segments. All the information that Chiapas knows about the specification is encapsulated within this structure. (See Figure B for a visual representation of this structure.)

Chiapas contains the chiapasManager application, which includes functionality for accessing and maintaining the Hierarchy Tree. This is covered in the chiapasManager manual, but it should be noted that this application can in fact create an entirely new specification or Hierarchy Tree from scratch if needed. Maintaining the Tree is covered further in the chiapasManager manual. In this chapter, we will examine the full definition of various building blocks of the Tree to lay a foundation of understanding with how Chiapas can interact with it to encode and decode business data.

The Hierarchy Tree represents every single loop, segment, and element defined by a specification, which in turn gives every valid choice that can occur within a HIPAA file defined by that specification. This includes valid values for elements, specific data types, limitations on the number of occurrences of loops and segments, and more. Chiapas often includes critical notes as given in the HIPAA Implementation Guides.

There are three primary building blocks of this Tree: loops, segments, and elements. They are detailed as follows:

Loops

A loop is the Chiapas representation of a loop as it is used in the HIGs; namely, a specific series of required and situational segments that may repeat itself. A loop is a grouping of similar pieces of information. For example, a service line loop will contain a number of segments about a single claim service line; a claim loop will contain all the information about a claim.

Except for the first Interchange loop, each loop is referenced as a “child” of an empty segment. This allows loops to be embedded between normal segments, such as between 'ST' (transaction start) and 'SE' (transaction end).

A loop has the following characteristics:

Name
Description
Repeats
Position
Segment Children

Name - The name is the short name for a loop, usually '1000A' or similar.

Description - Reflects the long name, as given in the HIGs, such as 'Member Name'.

Repeats - Refers to the maximum number this loop is allowed to repeat itself, with a -1 representing an infinite number of loops. A loop with a 1 in the Repeats field should occur once and only once within the context of the parent loop.

Position - The Position field identifies this loop in relation to other loops existing at a similar level. As an example, let's postulate that there are three required loops with positions 60, 70, and 70. The loop with 60 will always come first, but **either** of the two loops identified with 70 can occur in any order, as they are defined to inhabit the same “level” in the Tree. If a fourth required loop at position 71 were to be added, that could only occur after all instances of the loops at position 70 took place, and not before.

Segment Children - The Segment Children field is an in-order list of segments that comprise the loop itself. The first segment is the header segment, and identifies an iteration of that particular loop.

Segments

The segment can be thought of as an atomic unit of information within a loop. An N3 segment always defines the street portion of a postal address; a DTP segment always contains date/time information pertaining to some event. A segment is composed of elements, each element being a single unit of information.

Segments possess the following characteristics:

Name
Element Definitions
Required Flag
Repeats
Segment Identifier

Required Conditions
Position
Filter
Operation
Loop child

Name - The Name identifies the segment's business name, as given in the HIG.

Element Definitions – This is the pool by which elements are stored as part of a segment; elements are covered in the next section.

Required Flag – This indicates if it is a required segment within a loop. If this is the first segment within the loop, then the Required Flag tells whether the whole loop itself is required.

Repeats - The Repeats indicator tells how many times a single segment may repeat itself. A Claim Notes line, as one example, may occur several times to describe multiple lines of notes. Also, a segment that can contain a variety of different types of information (For example, different member ID numbers) and business requirements may need to convey all of these multiple identities using iterations of one segment.

Segment Identifier - This is the 2-3 characters that make up the base segment identifier. This would be 'CLM' for the Claim segment in 837, 'CLP' for the Claim segment in the 835 remittance advice, and so on.

Required Conditions - Many segments in the HIG contain a combination of required and situational elements. Sometimes elements are dependent on other elements under different conditions; if one occurs, then the other must occur. In this case, none of these are mandatory in the specification, but if one appears then the other must appear as well. Usually, these rules are spelled out with a letter, and two or more pairs of numbers identifying the elements subject to the rule. The HIG specifies many more requirement conditions than are actually given in the Chiapas Hierarchy Tree; this is done because in the early (4010) revisions of the HIGs, many of the specified requirement conditions match to all required elements, or unused elements. Only requirement conditions which can actually impact the data were reproduced in the Chiapas Tree.

Here are the different types of requirement conditions:

E - "Exclusive" - One and only one element must be present, never both.
R - "Required" - At least one of several elements must be present.
L - If the first element is present, then the second or third must be present as well.
P - If one element is present, then both must be present.
C - Either both indicated elements are present or none of them are.

Position - The Position number plays a similar role as the Position field in the Loops structure—it says in which order segments can appear within the loop. Segments at the same position can arrive in any order.

Filter - The Filter identifier provides a content-based trigger that distinguishes this segment from other segments with the same segment string in the same loop. Many times within a loop, there will be defined multiple REF (identification) or DTP (date/time) segments to give various pieces of information. Although they share the same three-digit Segment string, the types of information they convey and the qualifiers are unique within that loop.

A filter is defined as a number referring to an element position followed by an equal sign and a comma-delimited list of all possible values. For example, the filter '1=303,304,305' means "This filter is true whenever the values 303, 304 or 305 occupy the first element position."

One special case on this involves HI (Diagnosis) segments. In this case, the filter needs to reference the contents of a special case composite element, which itself is made up of a number of sub-elements separated by a special separator, often the "greater than" sign ('>'). In such cases however, there is a standard qualifier at the beginning of the composite element that can be addressed with the 'D' directive. The 'D' directive matches on the contents of the first element; for example, 'D=BK' means that the filter is true whenever the first element begins with the letters 'BK'.

Operation - This value is a special directive to the Chiapas Encoder. These directives adjust certain internal flags to make the Encoder behave in a certain way (depending on different circumstances) to make the Encoder easier to work with when making HIPAA files. There is more in-depth information on these directives in Chapter 6: Encoding Process.

Loop Child – Loops can be inserted between segments within a loop. Thus, the easiest way to represent it within a loop is as a segment containing no business data except for the "Loop Child" structure. This Loop Child contains all the information about the loop and also contains the segments within the loop. It is empty for a normal segment; if it has a value, then all the remainder of the information in the segment is ignored and it is treated as a loop.

Elements

The element is the structure that most closely corresponds to a data field of a given business system; therefore, all mappings refer to individual elements. There are also elements that play housekeeping roles within the HIPAA protocol and do not strictly encode information themselves. Often, elements occur in qualifier/identifier pairs when a segment can encode a number of different types of information within those slots.

Here are several types of elements:

General Data - A segment explicitly states one type of data for this element.

Segment Qualifier - This element qualifies the segment and contains either one value or one among a list of values for the possible qualifiers.

Data Qualifier/Identifier pair - This is a pair of elements wherein the value in the second element is predicated on the qualifier in the first element. In this case, the elements are said to be "overloaded" because they can have multiple meanings. In Chiapas, a reference can be made not only to a specific element, but also to a specific overload.

Composite - In certain circumstances, a single element will be split apart into multiple sub-elements, separated by the composite element separator given in the very beginning of the HIPAA file. In Chiapas, the element can be referenced whole, or referenced by each individual sub-element. Additionally, sub-elements may possess overloads.

An element contains the following values:

Name - The name represents the business field name most closely associated with this element.

Root Element - Within the Hierarchy Tree file is a table of short element names along with field type and min/max character lengths. All elements index to this table to define the basic information.

Outvalue - When populated, this value represents a default data value. A value in curly brackets will indicate a special encoding operation—for example, {DAT8} will emit an eight-digit current date. This will be explained further in Chapter 6: Encoding Process.

Values List - Some elements, especially qualifiers, can contain one value from a static list. Each value in the list defines three items of information: the name of the value, the actual value itself, and any notes associated with it.

Overloads - Overloads aid in mapping codependent element qualifier/identifier pairs. The overload represents the identifier, and when the overload is mapped instead of the element itself, a causal relationship is defined for this mapping. For decoding, this means it triggers when the qualifier is present, and when encoding, the qualifier is populated when the identifier is present.

Overloads are represented similar to elements, except that they possess Filter and Outvalue fields. The Filter and Outvalue fields on overloads normally hold the same value.

Sub-Element List - If this is a composite element, this is the list of sub-elements contained within the composite element. Sub-elements use the same data structure and may contain both Values Lists and Overloads themselves.

Required Flag - This is set when an element is required. If the segment is required and the mapping is missing, then Chiapas will automatically fill the element with spaces to the minimum size required by the root element.

Unused Flag - This marks the segment as Unused. It should never contain any actual data. When elements are displayed on the screen, marking an element unused allows the visual representation of the segment to match what is used in the HIGs.

Notes – Implementors can use this area to write notes about the mapping.

Chapter 4: Data Shaping Layer

Chapter Summary:
- Distinction between HIPAA values and Business Fields
- SQL as the preferred method to accomplish this
- SQLite's extra functionality for common conversion functions
- Data Shaping for Encode Operations
  - Enforcing Level Integrity
  - Ensuring Proper Field Format
  - Populating Static Fields
- AWK script

This chapter deals with the differences between valid HIPAA-compliant field information and data usable by business systems. When HIPAA was implemented, it was designed to be a compact protocol using the minimum number of characters required to convey certain information. This is often contrary to normal business data, where dates, diagnosis codes and other information are in the format native to the enterprise production software. Therefore, there is a step called “Data Shaping” that comes between the mapping layer and business layer.

Chiapas comes with a modified version of the public domain SQLite engine created by Dr. D Richard Hipp. In this customized version, several functions were added that are handy for healthcare operations; it is the recommended tool for data manipulation. It actually is not necessary to use SQLite for this purpose; if the implementor has the facilities to carry out these operations on the business system, then this layer can be handled entirely at that point. However, sometimes this flexibility is not available, so in order to present a truly turn-key solution, Chiapas provides the capability to do extensive data manipulation. Furthermore, all of these operations can be done by declaring a new 'in-memory' database. Since no insertion or deletion operations would need to hit a hard drive, these operations are very fast.

The 3.1.3 version of the SQLite engine has the following capabilities:

- Current time variables: CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP
- AUTOINCREMENT fields
- Indexing
- In-Memory
- Sub-Queries, Collated Sub-Queries
- Views
- Implements most of the ANSI-92 SQL standard
- The 3.1.3_rt version in Chiapas adds nine custom functions, as explained in Appendix C: SQLite Reference

Some items not implemented are foreign key constraints, ALTER table commands, RIGHT and FULL OUTER JOINS, and writing to Views.

Data Shaping from Decode Operations

Generally speaking, data from within a HIPAA file is normally very ”clean,” and usually most of it can be used as-is within a business system. There are several things to note, however:

- Chiapas can address only one entire element, sub-element or overload at a time. Because of this, 17-digit date ranges are returned in their entirety. In order to reduce the field to just the

'From' or 'To' part of the date range, the string must be extracted.

- Diagnosis codes in HIPAA are always represented in their “undotted” form. For business systems that require the dots, the to_dotcode function inside of the SQLite module will attempt to insert the dot back into the diagnosis code by the following rules:

No dot required:
- Three-digit diagnosis code, or a four-digit code beginning with an 'E'

Dot inserted:
- For diagnosis codes greater than three digits (four for E codes), the dot is inserted between the third and fourth characters (or between the fourth and fifth characters for E codes)

- Many business databases make a distinction between NULL values and a zero (‘0’). Sometimes zero values inside a HIPAA file will not be actually encoded, leaving it on the data shaping to make it a zero quantity.

- All date fields inside HIPAA are either six-character or eight-character fields (mostly the latter) in the formats YYMMDD or YYYYMMDD (year-month-day). If a database needs to put this result in a physical date/time field, it must first be converted.

Data Shaping for Encode Operations:

Raw business data can need a fair amount of data manipulation before it is ready to end up in a HIPAA file. It really depends on the day to day business processes and the quality of the data itself. In fact, a primary source for a HIPAA errors occurs when business data is suddenly corrupted, and data transport processes that have worked flawlessly for a long time suddenly break. Proper data shaping can mitigate this possibility, but having clean data is the first line of defense.

There are three categories of encoder-bound data shaping:
a. Enforcing Level Integrity
b. Ensuring proper field format
c. Populating static fields

a. Enforcing Level Integrity

Mappings in Chiapas occur at one or more levels. A level describes the number of layers containing unique data that are found in the data source. A file containing a fixed set of changes between records, with all other data being the same, can be described as having just one level. A typical claims file would be a two-level file, as the claim/member information would occupy one level and the service line information would be the next.

When Chiapas encodes data, the mapped fields are rearranged in a specific order (further explained in Chapter 6: Encoding Process) to ensure that the hierarchical HIPAA file produced matches the data within the flat table exactly. A product of this process is that any slight discrepancies in the source data for describing the same entity can result in dramatically different HIPAA files.

For example, let's say that Clinic A produced a file describing four visits throughout the month for one person, John Doe. Only one claim is generated, with each visit being on a different service line in the claim. On his fourth visit, John Doe supplies his last name as “John Dough.” Clinic A's

data system requires all the claim information to be reentered for each visit, even with the same claim number, and the new last name is entered in.

When the claim is brought into Chiapas, the fields are rearranged in a different order according to how the business fields map to the Hierarchy Tree. Since the name field occurs at a higher level in the hierarchy than the Claim ID, it is encoded first: “John Doe.” Then, the Claim information is encoded, and three service lines are encoded - all claim information is the same at that first level. The Encoder now goes to the fourth service line and compares the data with the previous line, and notes that the name is now “John Dough” instead of “John Doe”! To the Encoder, this is a signal that a new person is being described, so the current claim and member is closed out and a new member defined: “John Dough.” Then, the claim information is encoded again, and the fourth visit service line encoded.

In this scenario, because the data was non-normalized, a single discrepancy in the last name of the patient introduced a repeat of his member data and a repeat of the claim information itself, which is not HIPAA compliant. A business may decide to implement 100% clean data, but for most this is not practical. This is where data shaping comes in. The following assumes SQLite is used to do this.

There are several approaches to this problem, but in my experience a good approach is to use a self-join table reference based on a primary key and the ROWID field for each level embedded within the source table.

Example:

```
SELECT      T1.CLAIM_ID AS CLAIM_ID,
            T2.LINE_QTY AS LINE_QTY,
            T2.LINE_MODCD AS LINE_MODCD,
            T2.LINE_DOS AS LINE_DOS
FROM        ENC_837P_SFHP T2
INNER JOIN  ENC_837P_SFHP T1
ON          T1.ROWID = (SELECT MIN(ROWID) FROM ENC_837P_SFHP WHERE
CLAIM_ID = T2.CLAIM_ID)
ORDER BY    T1.CLAIM_ID, T2.LINE_ITEM;
```

This query would be appropriate for normalizing a two-level data source, where the two primary key fields are CLAIM_ID and LINE_ITEM. It utilizes the built-in SQLite variable 'ROWID', which is a numeric sequence starting at 1 for every new row inserted into a table. By matching on the smallest ROWID for any particular claim, then the 'T1' table represents the data from only the first table row associated with that particular claim. This is joined to the 'T2' table, which represents all of the level 2 data. The SELECT statements draw all level 1 data from 'T1' (tier one) and all level 2 data from 'T2'. In this way, the data for the primary claim information is fully normalized, and only the service line data is drawn straight from the data source. This data is unique on every row for a two-level data source, so this makes logical sense.

b. Ensuring Proper Field Format

This section is focused on turning business data into a form suitable for encoding into HIPAA. As discussed under Data Shaping from Decode Operations, number fields and date fields often need some conversion before they can be presented to the Chiapas Encoder layer. Also, data fields with unnecessary trailing or preceding space characters should be “trimmed.” Here is a broad overview of common data shaping functions as presented within SQLite.

Dates - Chiapas SQLite presents two functions, FX_DAT_TO_STR and FX_STR_TO_DAT, to translate a given date to and from the fixed format date string used in HIPAA. HIPAA uses YYYYMMDD exclusively to represent a date, in the form of year month day. These functions (explained further in Appendix C: SQLite Function Reference) can manipulate date and time information to and from this format.

Numbers - There are two commands, SOFT0 and HARD0, which deal with null information. In HIPAA, it is considered incorrect to convey a situational element field with a 0 quantity. For a required element in a mandatory segment, there must be a zero present. Furthermore, all preceding or trailing zeros (such as the '.00' in '500.00') are considered extraneous and should not be in the HIPAA file. The HARD0 SQLite function eliminates unnecessary zeros and emits a 0 if the input is a null string. The SOFT0 SQLite function also removes zeros from a numeric expression, except if the quantity is 0; it will then return a null value. Null mappings are not triggered by Chiapas, which means they will not be encoded and thus no unnecessary zero quantities will be created.

Strings - SQLite presents the TRIM function to shave preceding and trailing spaces from the expression. Spacing is generally only necessary in the outer interchange envelope when all fields must be a fixed size. In all other cases, strings should be represented with no extraneous spaces.

c. Populating Static Fields

Encoding requires much more information to be supplied than decoding. Some specifications require contact information for the submitter to be embedded in the file, and there are often specific business identifiers spelled out in the file that tie the submitted file to a well-defined business relationship between trading partners. Also, many values inside the outer envelopes must be supplied by the user. This all must occur at the data shaping layer, as the mapping layer and data source layer are kept separate in Chiapas.

The exact information that needs to be encoded varies between specifications and business requirements; more information on this is found on Chapter 6: Encoding Process.

AWK Scripting

The base Chiapas install includes an open-source text manipulation program called AWK. Used primarily in the Unix world, AWK is a powerful way to execute complex transformations on text files. An included sample AWK script transforms an incoming HCFA-1450 file into a comma-delimited values file that can be imported by Chiapas. This well known parser can be used to assist in data shaping.

Chapter 5: Mapping Layer

Chapter Summary:
Mapping Information
Segment Address
Element
Overloads
Special Cases
Extended Mapping Options
Decoding - Extended Mapping Filter
Encoding - Extended Overloading

Mapping entails making business decisions about what elements inside the HIPAA Implementation Guides best represent a set of business fields, and then communicating this to the HIPAA parser. Furthermore, each field must be associated with a level, which itself is attached to a unique primary key. The primary key fields play slightly different roles depending on the direction of the transaction, but they always play a critical role in managing the flow of information between the business system and a HIPAA file.

Each mapping contains the following information:

Order - A numeric index to the mapping. This starts at one and goes forward.
Level - A number representing the level for this field. Chiapas supports up to 20 levels, but even complex mappings can be done with just 2 levels. A negative number indicates that this field is a primary key. The number -1 means it is the primary key for level 1; -2 is the primary key field for level 2, and so on.
Mapping - A compilation of a segment address and an element or overload reference that resolves to a single element or overload.
Extended Filter - Optional information that provides flexibility for special business situations, such as custom overloading and filtering.
Notes - General information about the mapping.

A mapping consists of three parts:

```
[ Segment Address ] ! [ Element # ] ! <[optional overload or sub-
element #]> <! optional sub-element overload>
```

Segment Address

The segment address describes a path throughout the Tree. The first segment of each loop is what defines the loop, so therefore it is unique within that level. Therefore, by tracing a path of segments, a unique address can be made going directly to any segment within the hierarchy. It will consist of the two-character or three-character Segment identifier, and if this segment has a Segment filter, it will be included as well (in its entirety). It is then followed a colon (':'), and subsequently followed by the Segment of the next segment in the path. If the segment being mapped is not the header segment within the loop, then that segment is tacked onto the existing list.

Here is an example of a Segment address to the Rendering Provider License Number within an 837 Institutional Claim.

```
ISA:GS(8=004010X098A1,004010X098):ST(1=837):HL(3=20):HL(3=22):CLM:NM1(1
=82):REF(1=0B,1B,1C,1D,1G,1H,EI,G2,LU,N5,SY,X5)
```

The ISA segment does not need a Segment filter (it is alone at the top of the Hierarchy Tree), so the term 'ISA' followed by a colon starts off the mapping. After this level, there are a number of loops defining various specifications that occur under the ISA loop. In this example, we are describing a mapping within the 837I specification, so we next relay the segment header for that loop, the GS segment followed by the segment filter unique only to this loop. This pattern repeats itself, descending through several loops, until the REF segment is hit—and so now we can finish the mapping.

Element

As stated earlier, a mapping references a distinct element within a segment. In Chiapas mappings, after the segment address follows an exclamation and a number representing the element that is referenced. In our example, we would have a valid mapping with:

```
ISA:GS(8=004010X098A1,004010X098):ST(1=837):HL(3=20):HL(3=22):CLM:NM1(1
=82):REF(1=0B,1B,1C,1D,1G,1H,EI,G2,LU,N5,SY,X5)!2
```

This would map to the second element of this instance of a Rendering Provider ID segment. Note that this isn't explicitly to the Rendering Provider License Number. In fact, if there were multiple repeats of this REF segment outputting several different IDs, the mapping would probably just refer to the second element of the last REF segment to occur, whichever it may be. Obviously, this is not specific enough for use in a business. Fortunately, we have overloads.

Overloads represent the mapping relationship on qualifier/identifier pairs inside HIPAA. In the case of the REF segment above, this single segment could convey twelve different types of ID about the Rendering Provider. The first element qualifies the type of ID, and the second element is the actual ID. Inside the Hierarchy Tree, the first element will list a number of valid values used for decode validation, and the second element will contain a list of twelve overloads for each of the IDs. In the mapping, this overload is referenced after the element # with another '!' mark and the filter for the overload, which for this mapping is '1=0B.'

Therefore, the full mapping in our example would look like:

```
ISA:GS(8=004010X098A1,004010X098):ST(1=837):HL(3=20):HL(3=22):CLM:NM1(1
=82):REF(1=0B,1B,1C,1D,1G,1H,EI,G2,LU,N5,SY,X5)!2!1=0B
```

Special Cases

There are two special cases.

The first special case deals with “floating” overloads found in PER segments. PER segments contain data about contact information such as cell phone numbers or e-mail addresses. A PER segment contains three slots for a qualifier/identifier pair that can be one of several different items, such as telephone number, extension, etc. Technically, HIPAA allows this information to be valid in any one of those three slots. Therefore, in lieu of an element number, which may be wrong, it is possible to substitute with one of six two-letter contact type identifiers:

```
EM - E-mail
FX - Fax Number
TE - Telephone Number
EX - Telephone Extension
HP - Home Phone
WP - Work Phone
```

The 'EM' element identifier will map to the e-mail identifier, no matter which of the three qualifier/identifier slots it resides in.

The second special case deals with sub-elements and sub-element overloads. As the HIPAA hierarchy sometimes defines elements that are split into sub-elements, then these need to be addressable via mapping as well. Also, they may have overloads that apply just to that sub-element frame of reference.

The format for a sub-element reference is to specify the element, another '!' mark, and then the sub-element number. To address a sub-element overload, follow it with yet another '!' mark and then the filter string for that overload, similar to this example:

```
[segment address]!2!2!1=AB
```

This would address the second element and the second sub-element within that element, where the first sub-element is 'AB.'

Extended Mapping Options

In addition to the above, there are two extensions to normal mappings to deal with special situations during the encoding and decoding. These are not part of the normal mapping string, but exist in the "Extended Filter" section on each mapping. The extended mapping filter is split into two parts separated by a semicolon. Information preceding the semicolon deal with extended decoding filters; any information present after the semicolon defines an extended overload.

Decoding - Extended Mapping Filter:

The HIPAA format allows multiple iterations of some loops. These loops may exist for some business records but not others, making them unsuitable to be mapped with primary keys. Also, different iterations may play different business roles, making it more suitable to map them to specific columns as opposed to multiple rows. Therefore, each mapping may contain extended filtering options that will trigger the mapping only when a specific iteration of a loop or segment is encountered.

The exact iterations of each loop are stored internally by the Chiapas Decoder as it progresses through a HIPAA file. This is tracked in an array that assigns ten numbers to each segment. the first nine numbers represent nine loop nesting levels, with the ISA / IEA segments representing the very first nest level. Therefore, the nest array contains ten 1s ('1111111111') for the ISA segment; it begins with a 2 for the first nest level for the entire HIPAA file (except for the IEA segment, which is a 3 followed by nine 1s, or '3111111111'). The 10th nest level represents segment repeats. It is a 2 for the second iteration of a segment, 3 for the third, and so on.

To individually address these iterations, there is a special syntax for the extended mapping filter. 'L01' matches to the first nest level; 'L10' matches to the tenth, which is the segment repetition

counter. Therefore, placing 'L10=2' will match on the second occurrence of a given segment within the loop. Knowing the exact numbers to match a segment on within a structure may be difficult. Therefore, every DECODE operation done by the Decoder changes the loaded HIPAA object to include the array filter within each segment. By doing a DECODE operation and then viewing the HIPAA object, it is possible to easily see the exact indexing for each segment. In turn, this allows the user to create extended mappings specific to the business rules.

More information on the iteration array will be presented in Chapter 7: Decoding Process.

Encoding - Extended Overloading:

There are some segments within the HIPAA hierarchy where a single, predefined set of overloads is not practical. When there are multiple qualifier fields, each having a large number of valid values, it is impossible to create an overload to handle every qualifier/identifier scenario. This is where extended overloading mappings come in.

An extended overload acts just like an overload outvalue string, except that it can be custom defined for each mapping. For example, in the 835 Remittance Advice specification, there is a segment for emitting claim adjustments, 'CAS.' There is both a Claim Adjustment Group Code that identifies the broad business reason for the adjustment, as well as a second element giving a specific reason code. A claim may have a number of different Claim Adjustments, but there are no overloads that handle all combinations of these two elements without manually adding one to the Hierarchy Tree.

Therefore, it's possible to add an extended overload in the extended filter field. As it shares the same field as the extended mapping filter, a semicolon (';')must precede the extended overload. Afterwards, it can follow the normal syntax of any overload, and this will fire whenever there is data to encode for this field, in addition to any overload selected in the primary mapping.

Chapter 6: Encoding Process

Chapter Summary:
- How it Works
- Encoder Initialization
    - 1. Mappings Validation Check
    - 2. Refine Data Key and Mapping Key
    - 3. Verify Source and Destination columns are the same
    - 4. Create a new Encoder instance
        - a. Initialize ascending fields
        - b. Create a sorted data key
    - 5. Set the Seed Transaction Numbers from mappings
    - 6. Create a new SegPool object
    - 7. Iterate Hierarchy Tree
    - 8. Return SegPool object
- Using the Encoder
    - Required Elements for all HIPAA transactions and overriding Hierarchy Directives
    - Common Encoding Tasks
    - Troubleshooting Common Problems

To understand the process of encoding business data into a HIPAA file, it is necessary to have a solid understanding of the Hierarchy Tree, Data Shaping, Mapping, and finally, an understanding of the behavior of the Encoder itself. This chapter explains the inner workings of the Chiapas Encoder and how it can be used to deal with a number of business scenarios. First, I will discuss how the HIPAA Encoder initializes itself, traverses the Hierarchy Tree and pulls data from the data source defined by the mappings. Then, I will discuss the outer envelopes common to all HIPAA files and their requirements. Afterward, I will discuss the special Encoder directives placed within the Hierarchy Tree to deal with certain situations.

SECTION 1: How the Chiapas Encoder works

Encoder Initialization

1. Mappings Validation Check
2. Refine Data Key and Mapping Key
3. Verify source and destination columns are the same
4. Create a new Encoder instance
    a. Initialize Ascending Fields
    b. Create a sorted Data Key
1. Set the Seed Transaction Numbers from mappings
2. Create a new SegPool object
3. Iterate Hierarchy Tree
4. Return SegPool object

1. Mappings Validation Check

In this phase, the incoming Mapping Key undergoes a series of checks to make sure it is valid for encoding:

1. Is the Mapping Key empty?
2. Are the key levels within a range of 1 through 20?

3 Are the primary key levels within -1 through -20?
4 Is there only one primary key field per level?
5 Are the mappings named properly? It must start with an alpha character and may only contain alpha characters, numbers, and the underscore character ('_')
6 If the key does not have a blank mapping value, it must resolve to a valid reference within the Hierarchy Tree.
7 Any keys beginning with 'CHIAPAS_' are culled from the Mapping Key.

2. Refine Data Key and Mapping Key

In this phase, the incoming data and mappings are culled of unused information (mappings without an element reference).

3. Verify Data Key and Mapping Key Match

At this point, the number of columns in the data source is matched with the number of active mappings in the Mapping Key. The field names do not have to match with the mapping names; only their numeric order is referenced. Any mismatch aborts encoding at this point.

4. Create a new Encoder Instance

This a data structure used internally by the Encoder to keep track of different information; how it is initialized and set up play a large role in the behavior of actual encoding.

a. Initialize Ascending Fields

It is important to know exactly how the Encoder is going to be traversing the Hierarchy Tree at this point to gain an understanding of how this initialization works. The Hierarchy Tree itself possesses a universal “depth” number for each segment, element, sub-element and overload, beginning with '1' for the very first ISA segment and then increasing. The last addressable IEA element would thus have the greatest depth.

This depth plays a critical role in the sorting process, and in the way mappings translate a normalized data source into a hierarchical data file. The mappings are grouped and sorted by their relative depth, with the least-depth mappings occurring first and the biggest-depth mappings occurring last on a one-dimensional (“1D”) array.

Each mapping on this 1D sorted array represents an “Encoder target.” The Encoder will mark a path to the first mapped target from the topmost segment all the way down to that target. In most cases, the first few mappings will be in the ISA segment itself in order to display required envelope information. As the Encoder hits these targets, it maps out a path to the next target, and will follow situational loops and segments to reach it. Then, it will hit this target and continue on to the next one (more about how this occurs is in a later section). For right now, it's important to know that this 1D array of fields represent a sequentially ordered list of targets for the Encoder.

Basically, the Encoder works in a sequential order, from left to right on this target array. The exception to this rule pertains to repeated mappings of the same elements in the same segment. Chiapas will group both of these independently of their strict depth-order mapping in order to make sure the Encoder handles discrete segments sequentially.

After the 1D mapping array is created following these rules, then Chiapas sorts the Data Key based upon the primary key fields. The level 1 primary key field has first order sorting priority, level 2 primary key field second order sorting priority, and so on.

5. Set the Seed Transaction Numbers from mappings

Once the data is sorted and the 1D Mapping Array is created, the Mapping Key is scanned for seed interchange identification numbers. These affect the unique transaction identifiers found in the envelopes of the HIPAA transaction.

The keys themselves are:

`CHIAPAS_SEED` – This number simultaneously sets the ICN, the GCN and the TCN. The GCN is the ICN plus one; the TCN is the ICN plus two.
`CHIAPAS_ICN` – Individually sets the ICN (interchange control number)
`CHIAPAS_GCN` – Individually sets the GCN (group control number)
`CHIAPAS_TCN` – Individually sets the TCN (transaction control number)

6. Create a new SegPool Object

At this point, a new blank SegPool is created. This is the object that will be populated by the Encoder and returned.

7. Iterate Hierarchy Tree

At this point, the control IDs have been set, the source data is sorted, and the Mapping Array has been sorted and grouped by segment. All of the ingredients are present to start the encoding itself, except for the thresholds. Thresholds subdivide the mappings according to their levels, and identify locations within the 1D Mapping Array that define the level boundaries. Once these are defined, encoding begins.

The Encoder starts with the leftmost entry for the Mapping Array and sets that as a target. It also starts with the very first segment in the Hierarchy Tree, which is the ISA segment. A pointer is set to point to the first data row, which contains the lowest-valued array sorted items, and traversal of the Hierarchy Tree begins

The Encoder will descend into the Tree, moving to the next mapping after each target is hit. If there is no data in the target, it is skipped and the next target with data is set instead. The encoding continues with the Mapping Array pointer advancing to the right and extracting data from the Data Key and emitting it to segments. This continues until the Mapping Array pointer goes too far to the right, and the Encoder decides it's time to advance to the next row. When that happens, the current row is compared to the next one, and the Encoder makes a decision of where the Mapping Array pointer should be set.

Encoding continues until the entire data source is exhausted of data, and then the Hierarchy Tree exits from the given specification and emits the envelope ending segments.

8. Return SegPool Object

Now that the Encoder has created the encoding object, it is returned to the caller.

SECTION 2: Using the Chiapas Encoder

1. Required Elements for all HIPAA transactions and overriding Hierarchy Directives

a. ISA segment

The ISA segment represents the start of any X12 file, and lays out some fundamental data about the file. The ISA segment:

- Defines the basic syntax of the file itself. The first segment has fixed length fields to lay out the segment, element, and sub-element separators.
- Identifies the senders and receivers for the file. Only a very basic 15-character identifier is allowed here.
- Identifies the date and time the file was generated, as well as if this identify if is a production or test file.

  This segment contains many default values, as described in Chapter 3. All of these can be overridden using explicit mappings if necessary.

Example:
```
ISA*00*??????????*00*??????????*ZZ*XY55000????????*ZZ*00999??????????*0
50223*0309*U*00401*100400006*0*P*>~
```

The first two present authorization and security information—they are left blank in this segment. The 'ZZ*XY55000' information represents information about the Sending entity. The next element pair ('ZZ*00999') gives information on the Receiver entity. '050223' is the date the file was created in YYMMDD format, and '0309' is the time (3:09 AM) in 24-hour format. '00401' is a version number, and the '100400006' represents an Interchange Control Number, or ICN. The ICN should be supplied via a CHIAPAS_SEED mapping externally, or mapped explicitly, but it definitely should be set prior to encoding. It represents an ID to uniquely identify this file.

The '0' means that no Acknowledgment is requested. This is a short file generated by the recipient that guarantees the file was successfully received and decoded. A '1' means that a Functional Acknowledgment is being requested. Chiapas has the 997 Functional Acknowledgment specification built in, and Functional Acknowledgments can be processed like any other HIPAA file. More is covered on this in Chapter 7: Decoding Process.

b. GS segment

The GS segment is the start of the Functional Group envelope. This is segment identifies which version of which specification is used to emit the data. It also gives another sender and receiver ID.

Example:
```
GS*HC*XY55000*00999*20050223*0309*100400007*X*004010X096A1~
```
In this sample GS segment, the 'HC' identifies this is an 837 claim envelope, from someone with sender ID 'XY55000' to someone with the Receiver ID '00999,' on 02/23/2005, at 3:09 AM. The '100400007' identifies the Group Control Number (GCN), an ID used to describe this particular Group envelope.

If the CHIAPAS_SEED value is set during mapping, this will automatically be set to the Interchange Control Number plus one.

c. ST segment

The ST segment starts the actual transaction containing the data itself. It should be noted that the containing envelope structure, the GS/GE loop, may contain multiple ST/SE loops. For example, in the 835 Remittance Advice format, a single ST/SE loop allows you to define only one check at a time. Therefore, to describe multiple checks of financial activity within one file, you would need to allow for several ST/SE loops.

There are only two elements, the specification identifier and a unique Transaction Control Number (TCN). The specification identifier is defaulted via element directives, and the TCN is normally the CHIAPAS_SEED value plus two.

Example:
```
ST*837*100400008~
```
This defines an 837 transaction with a unique TCN of 100400008.

d. SE segment

When all segments within a transaction have been sent, an SE segment “closes” the transaction loop. It consists of the number of segments (inclusive of ST and SE), and a repeat of the TCN.

Example:
```
SE*30*100400008~
```
This SE segment indicates the transaction loop was 28 segments long, plus the ST and SE segments, with the TCN of 100400008.

e. GE segment

The GE segment closes out the GS loop. It stores the number of transaction sets and a repeat of the GCN.

Example:
```
GE*1*100400007~
```

This closes out a group containing one ST/SE loop with a Group Control Number of 100400007.

f. IEA segment

This gives the number of groups and a repeat of the ICN.

Example:
```
IEA*1*100400006~
```

This indicates one group was sent with an ID of 100400006.

2. Common Encoding Tasks

Optional Loops:

The Encoder will not encode mappings that do not possess data. So, if all of the mappings for a situational loop contain null data, that loop will be entirely skipped. If the loop is mandatory, then the Encoder will descend into it and emit invalid segments with null information.

Therefore, strict data shaping must be done beforehand on the mappings pointing to optional loop information items to set them to NULL before encoding.

PER segments

When decoding contact information from a PER segment, you can use shortcut names like 'EM' or 'HP' (for Email and Home Phone) to simplify accessing the floating information. However, when encoding, you must explicitly give the overloads for the information.

HL segments

This segment defines four elements that ascribe a number to looped information within the HIPAA file. Commonly, each subscriber loop exists as another HL* loop level within an 837 claim file. The fourth element is a '0' or '1,' and indicates whether there is a child HL* loop. These are defaulted by Chiapas, but care should be taken for situational child loops where Chiapas will give the incorrect default ('0' instead of '1'). Mapping this and setting it to '1' when there exists situational child loop data will correct this.

NM1 segments

These segments describe names. Sometimes they will communicate corporate and personal name information; Chiapas provides overloads in these cases to map these correctly.

DTP segments

All date and time information communicated by these segments must be in the proper format for dates used by HIPAA, often YYYYMMDD.

CLM segments:

These segments possess a composite element in the Type of Bill code. The entire segment may be constructed via three individual mappings to the constituent parts, or a value passed in for the whole element.

HI segments:

Each element is a composite element—that is, composed of a number of sub-elements and sub-element overloads.

CAS segments:

Claim Adjustment segments exist in 835 Remittance Advice and 837 Claims files, and exist to give specific reasons behind a numerical adjustment to a claim or line amount.

Extended mapping options need to be used in order to provide default reasons for specific business adjustments.

Duplicate Mappings

Chiapas groups element mappings in such a way that a repeat of one mapping will result in a new iteration of the entire segment—it does not overwrite the value of that same mapping. This is by design in order to allow for multiple instances of a single segment with different information (for example, a claim note segment).

If the duplicate mapping occurs inside of a loop header segment, then that segment and ALL required child loops, without information, will be emitted as well

Non-Normalized Data

The Chiapas Encoder absolutely requires normalized data. Normally this requires a bit of data shaping, as explained in Chapter 4: Data Shaping, to ensure that all of the data associated with a high-order mapping stays static throughout iterations of low-order mapping data.

Otherwise, Chiapas will not anchor the Mapping Array pointer to only the information within the low-order mapping group. It will instead try to encode it as new data.

As an example of this problem, we can delve into a problem I encountered when creating the 997 Mapping. During every decode operation, a functional acknowledgment key is created that is paired to the 997 Mapping Key – this way, creating the Functional Acknowledgment is as simple as feeding the Acknowledgment Key and the 997 Mapping Key to the Chiapas encoder.

But first, a problem. The ST02 and SE02 identifiers are stored so that they could be emitted as AK2 segments in the 997. But, I specified the SE02 – which isn't necessary actually – as a Level 1 mapping instead of a Level 2 mapping. This meant that the encoder created a new ST/SE loop in the destination acknowledgment file for every ST/SE loop in the source file – which is not correct. By omitting the SE02 information, only one element (ST02) had the primary key information and then the functional acknowledgments were correct.

Chapter 7: Decoding Process

Chapter Summary:
- Populate Hierarchy Tree with Mappings
- Initialize the Data Accumulator
- Descend into Hierarchy Tree
- Consume segments as they are encountered within the Hierarchy Tree
- When a hit occurs between the Hierarchy Tree and the next segment to be consumed, store the values within the elements back to the Data Key being generated
- When a hit occurs with highest order Primary Key mapping, set a latch. When the depth of the current hit decreases with the latch set, this is the signal to create a new data row.
- Return the created Data Key
- Functional Acknowledgments

Decoding a HIPAA file acts much like a reverse mirror image of the encoding process. The HIPAA file is loaded, split into segments, and then the Decoder "consumes" them as it finds matches between the Hierarchy Tree and the segment. If the elements in the Hierarchy correspond to mappings, then data is stored to an outgoing Data Key. When a highest-order primary key mapping is encountered, a new row is triggered. Once the file is completely consumed, the Data Key is returned.

1. Populate Hierarchy Tree with Mappings

The Mapping Key is loaded and references to the mappings are stored directly within the Hierarchy Tree. When the Decoder uses this Tree to consume segments from the incoming HIPAA file, these mappings will send data to the Data Key.

2. Initialize the Data Accumulator

The Data Accumulator is an internal structure used for decoding. It contains a copy of a new, blank Data Key as well as information about the conditions by which a new row is created.

3. Descend into Hierarchy Tree

The Decoder begins with the outermost segment, ISA, and compares it with the first segment from the incoming HIPAA file. If it is not an ISA segment, then decoding will stop here. Assuming it is, then the ISA segment will be consumed, and then the Decoder will try to find the GS segment by checking various situational GS/GE loops underneath the ISA loop. The "Position" defined for each segment and loop drives this process, as items with the same position can occur in any order. Therefore, this means the Decoder must do some backtracking to ensure that every segment is identified properly.

If a segment consumed contains mappings, then data is propagated to the current Data Row Key. This continues until the element corresponding with the highest order primary key is found, when a flag is set. This flag says that when the Decoder encounters that segment again or anything higher in the Hierarchy Tree, that the current data row is "complete" and will be added to the finished Data Key.

Segments continue to be consumed until either the Decoder cannot resolve a segment, or they are all consumed. As the SegPool object is consumed, it is also altered to reflect the ten-number identifiers assigned to each segment. This can be reviewed by a developer to create mappings

tied to specific iterations of a loop or segment. Also, an “Acknowledgment Key” is being created that allows a Functional Acknowledgment file to be created very easily.

4. Return the Created Data Key

After all segments are consumed, an error check is made to verify that the segments that were decoded equals the segments that were initially brought in. If there is a mismatch, then a segment was encountered that did not follow the specification and thus breaks the decoding process, and a critical error is returned.

Otherwise, the newly completed Data Key is returned.

5. Functional Acknowledgments

When decoding is completed, an Acknowledgment Key (“Ack Key”) is created. This is a Data Key possessing one row for every ST/SE loop that was decoded, and ties directly to a “997 Map Key” supplied with the Chiapas installation. When the Ack Key is fed to the Encoder along with this 997 Map Key, a Functional Acknowledgment file will be created relating the decode status of the file: acceptance or rejection. In some trading partner agreements, these files are required as a receipt of acceptance.

Chapter 8: JobScripts

Chapter Summary:
- JobScript exposes the functionality of the chiapasCore engine to a straightforward batch scripting system.
- Usage of different commands in a functional environment
- Setting up a mini-clearinghouse using “drop in files” via JobScripts.

Chiapas exposes the functionality of its various facets via a simple scripting system called JobScript. A “Job” in this case is a recurring business task, and “Script” refers to the simplistic syntax of the language used to communicate instructions to Chiapas. There is a chapter covering every command. This overview of scripting is focused on creating and using JobScripts to achieve business aims.

A JobScript's scope is to encapsulate functionality directly related to processing HIPAA files; this can include error handling, file moving, maintaining process logs, and more. Towards this scope, JobScript contains several dozen commands relating to a broad spectrum of functionality.

JobScripts can be executed three ways: Shell, Polling, or API.

Execution of JobScripts via Shell

Chiapas allows users to launch an interactive shell. Two commands available in this shell are:

```
PARSE_FILE_OLDSTATE [script name]

PARSE_FILE_NEWSTATE [script name]
```

Both of them will load the given script and execute it. The first one will execute the script with the existing JobState, meaning all variables will be the same. The second one will create a new JobState and pass variables 99 and 100 from the old one to the new one, execute the script, and then propagate variables 99 and 100 back to the caller's JobState.

Execution of JobScripts via Polling

When Chiapas is launched as a “daemon,” it scans certain directories for scripts every second. These directories are off the Chiapas Root directory and they are called:

```
SCRIPTS_PENDING

SCRIPTS_PERM
```

Any script placed in the SCRIPTS_PENDING directory will be processed and then placed into the SCRIPTS_COMPLETE directory. Scripts put into the SCRIPTS_PERM directory will be executed repeatedly; these are best for scripts that check for the existence of files in an incoming bin directory.

Execution of JobScripts via API

When the chiapasCore.dll module is linked via a .NET framework or Mono-compatible development system, scripts can be executed via code. For this to happen, a JobState structure needs to be created; its path needs to be set; and whatever variables the script is depending on

for input need to be set to the strVar string array. Once this is done, it can be called via the 'chiapasJobs.ProcessJobFromString' execution call. It takes two arguments: a string representing the entire script, and the JobState structure.

When execution completes, it returns a log of every statement executed. This log can be checked to verify that every statement executed properly.

Example of executing a script via VB.NET:

```
Dim Jobs As New chiapasJobs.JobState
Jobs.strPath = "base path"
Jobs.strVar(0) = "data..."
Jobs.strVar(1) = "data..."
Dim strError As String = chiapasJobs.ProcessJobFromString(strScript,
Jobs)
```

Syntax of a JobScript

JobScript is a relatively simple scripting language; it should be compared to DOS Batch files. It offers many options, including simple looping structures, the ability to execute SQL commands and query SQL databases. Many of these commands can be executed via the Shell interface, allowing users to create code and test it outside of the scripting environment. Looping, Conditionals and database connection commands can only be called inside of a script (with one exception, the PARSE command). Most of a script's responsibility will be to call upon the ENCODE or DECODE commands, but there are prerequisites to these actions. First and foremost, many commands inside JobScript are dependent upon state-dependent variables. Every time a JobScript runs, a new internal structure called a JobState is created.

Script Variables

A JobState stores all of the variables contained within the JobScript. These are:

Error Log – A list of all error messages from the Encoder and Decoder. After decoding, all HIPAA formatting errors encountered will be listed here.

SQL Connection – The currently set connection to a SQL database.

DB Type – The current database type.

Data Key – A representation of a two dimensional grid of business data, similar to a SQL database table without data types or field lengths. Data Keys are the result of a decode operation and part of the input for an encoding operation.

Map Key – A series of business field names attached to points within a Hierarchy Tree. The Mapping Key also defines levels and one primary key mapping for each level; these help define the business data fields in relation to the hierarchical data structure of HIPAA.

Ack Key – After every successful decode operation, a new Acknowledgment Key is generated. Many businesses require a "receipt of acceptance" from recipients as proof the data system successfully parsed the incoming HIPAA file. When this Ack Key is used as a Data Key and combined with a 997 Mapping Key, an encode operation will create a new Functional Acknowledgment HIPAA file that can be returned to the sender as proof of acceptance.

SegPool – This is an internal representation of a raw HIPAA file. It contains extended formatting information such as CR/LF after segment, 80-column split, and separator characters.

Tree – This is an internal representation of all of the data associated with the HIPAA Implementation Guides. It needs to be reloaded before every decode or encode operation.

ICNSeed – When set, this sets the baseline value for the Interchange Control Number used for encoding operations. This serves as a unique identifier within HIPAA transactions. Two other unique identifiers, the Group Control Number (GCN) and Transaction Control Number (TCN), are derived automatically when this is set. The GCN is the ICN plus one, with the TCN being the ICN plus two.

Path – This represents the base path of the Chiapas installation.

Script Variables – JobScripts allow for 100 internal variables, numbered 1 through 100. Each variable can contain a string of characters.

JobScript commands may be preceded by tabs or white space. After the command itself, one or more arguments may follow separated by the vertical bar character ('|'). This character is seldom used inside the arguments in JobScript commands, so it makes a good separator.

Script Variables are accessed via a number 1 through 100, surrounded by two dollar signs ($$) on either side. For example, any expression referring to '$$100$$$1$$' would resolve to variable 100 followed by variable 1. Many internal JobState variables are accessed as a name surrounded by double dollar signs, such as the path $$PATH$$. (See Appendix A: JobScript Reference for specifics on all of the variables accessible in this way.)

When the script parser is expecting a string or a number aside from a variable name, then variable substitution can take place in the expression. Following this is an example of a short JobScript using variable substitution.

Sample Script

```
TREE_FROM_FILE $$PATH$$\TREE\tree_exp.bin
MAPKEY_FROM_FILE $$PATH$$\MAPKEYS\MAP_BIZ1_BIZ2.key
DATAKEY_FROM_CSV_HEADER $$PATH$$\BIZ2\STAGE03.CSV
ENCODE
HIPAA_TO_FILE $$PATH$$\BIZ2\RESULT.TXT
```

The $$PATH$$ referenced in the file paths resolves out to the base path of the Chiapas installation, meaning the \TREE and \MAPKEYS directories are subdirectories off the main directory.

These five commands represent the minimum necessary to encode a HIPAA file and store it to disk. For every encode operation, there are three minimum requisites:
- Load the Hierarchy Tree
- Load a valid Mapping Key
- Load a Data Key that corresponds to the Mapping Key

Once this is done, the ENCODE command will execute the encoding process and store the result in the SegPool variable. The command HIPAA_TO_FILE stores this object to a physical HIPAA

file on disk.

Loops and Conditional Operations

A script may need to process multiple incoming HIPAA files, or process rows in a database. Also, it may need to take different logic paths depending on if a file is successfully loaded or not – like moving errored files to an error bin.

Loops

JobScript supplies loop functionality via the @FOR / @ENDFOR statements. There are several types of @FOR commands that allow for iterating through SQL rows, numeric sequences, text file lines, or directory contents.

The syntax on a @FOR command is:

@FOR [variable name] | [loop type] | [loop type dependent arguments]

The Variable name is a number 1 through 100. As the @FOR loop iterates through the values, the result is stored to this variable (referenced via $$1$$ through $$100$$). The loop type indicates the type of @FOR loop; in our sample script above, 'DIR' will indicate this will iterate through the contents of a directory. The third argument depends on the loop type; in our sample script, it is the name of the directory.

Internally, Chiapas will fully resolve the results of the directory and store them internally. The first result will be stored to the given variable, and all statements between the @FOR and @ENDFOR will be executed. Then, these statements will be repeated for all possible values for the loop.

Building on the previous sample script, let us change the goal from processing one specific file under the Chiapas path (`BIZ2\stage03.csv`) to instead process all CSV files within that directory. To do this, we'll need to add in a loop statement.

```
@FOR 1|DIR|$$PATH$$\BIZ2
	TREE_FROM_FILE $$PATH$$\TREE\tree_exp.bin
	MAPKEY_FROM_FILE $$PATH$$\MAPKEYS\MAP_BIZ1_BIZ2.key
	DATAKEY_FROM_CSV_HEADER $$1$$
	ENCODE
	HIPAA_TO_FILE $$PATH$$\PROCESSED\HIPAA_$$1$$
@ENDFOR
```

First, note the addition of TAB characters – this helps organize the script code and makes it easier to read, which cuts down on mistakes.

Second, the '1' specified in the @FOR statement is referenced in the DATAKEY_FROM_CSV_HEADER command. This assumes that every file within this directory is a Comma-Separated Value file with a one-row header. After the ENCODE operation is complete, the result is stored to a file inside the PROCESSED directory underneath the root path, with the original filename prefixed by the 'HIPAA_' string. Note that this script does not move the original files out of the BIZ2 directory – it is a simple example of the looping construct and is not necessarily a production-quality script.

Conditional Statements

Chiapas includes conditional branching statements – namely the @IF/@ELSE/@ENDIF statements. The syntax of the @IF statement follows:

@IF [expression 1] | [= OR < OR > OR <> OR <= OR >=] | [expression 2]

…

@ELSE

…

@ENDIF

The @IF command establishes a conditional branch depending on the comparison of expression 1 and expression 2. If the Boolean expression is true, the section from @IF…@ELSE (or @IF… @ENDIF if @ELSE is not present) is evaluated. If the @ELSE expression is present, the @ELSE…@ENDIF is evaluated instead.

This can be very useful for handling critical errors. The Chiapas variable $$ERRCRIT$$ contains a '1' when a critical decode error occurs, meaning the file could not be successfully decoded. In this code snippet example, this is checked to decide where to put a source HIPAA file:

```
@IF $$ERR_CRIT$$|=|0
      FILENAME 1|3
      MOVE $$1$$|$$PATH$$\COMPLETE\HIPAA_$$3$$
@ELSE
      FILENAME 1|3
      MOVE $$1$$|$$PATH$$\ERROR\$$3$$
@ENDIF
```

Using Data Keys

Data Keys are Chiapas' internal representation for a body of data. Data Keys store one or more Data Rows; each Data Row contains a fixed number of columns, consisting of the column name and the item of data. Data Keys are the result of a DECODE operation, and a source for an ENCODE operation. Since Data Keys play such a central role inside Chiapas, there are a number of commands relating to retrieving data from external systems, as well as storing Data Keys. Following is an overview of different transport methods involving Data Keys.

CSV Files

DATAKEY_FROM_CSV_NOHEADER [CSV file name]
DATAKEY_FROM_CSV_HEADER [CSV file name]
DATAKEY_TO_CSV_NOHEADER [CSV file name]
DATAKEY_TO_CSV_HEADER [CSV file name]

These commands load and store Comma-Separated Value files. Fields will be stripped of commas and double quotes are converted to single quotes before storing to a CSV file. When CSV files are loaded with a header, field names are loaded from the initial header row. If there is no header row, field names are defaulted to Field01, Field02, and so on.

Flat Files

DATAKEY_FROM_FLAT [format file name] | [flat file name]
DATAKEY_TO_FLAT [format file name] | [flat file name]

These commands provide an interface to fixed length flat files. Format files are two column Comma-Separated Value files indicating column length and column name.

SQL Tables

DATAKEY_TO_SQLTABLE [table name]
DATAKEY_FROM_SQLTABLE [table name]

These commands require an active, open SQL connection to a SQL database. When the DATAKEY_TO_SQLTABLE command is called, it will attempt to insert all rows of the Data Key to the given table name. If it does not exist, this will result in a SQL error. One exception to this rule is for SQL Server and SQLite, where it will create the table on the target server first if it does not exist. Every column will be instanced as a 200-length VARCHAR or TEXT field in a new table.

For non-SQL Server/SQLite databases, it is possible to run a SQL_EXECUTE command to create the table.

On the second command, the entire table will be transferred to the Data Key. Every table row will be a new Data Row.

Database Commands

Chiapas emphasizes a “black box” approach to an EDI solution; it is a capable, robust middle tier that can enter into a business system with a minimum of development on the surrounding systems. As many business systems use large databases to handle data, it is necessary that Chiapas provide some database commands in order to interact with them.

Opening and Closing Databases

```
DBTYPE [database type]
CNN_OPEN [connection string]
CNN_CLOSE
```

The Database Type needs to be set whenever a database connection is opened with the CNN_OPEN command. Databases are opened via a Connection String. Many examples can be found on the public website *www.connectionstrings.com*
Here are also some examples:

SQLSERVER:

Trusted Connection:
```
Server=Test1;Database=pubs;Trusted_Connection=True;
```

Passworded Connection:
```
Data Source=Test1;Initial Catalog=pubs;User Id=sa;Password=asdasd;
```

Via IP Address:

```
Data Source=101.102.103.104,1433;Network Library=DBMSSOCN;Initial
Catalog=pubs;User ID=sa;Password=asdasd;
```

ORACLE:

```
Data Source=MyOracleDB;Integrated Security=yes;
Data Source=MyOracleDB;User Id=username;Password=passwd;Integrated
Security=no;
```

SQLITE:

In-Memory Database (Cleared for every connection):

```
Data Source=:memory:;New=True;Version=3;
```

Disk Database:

```
Data Source=C:\EDI\mydb.db;New=False;Version=3;
```

ODBC:

These connection strings vary according to the ODBC database provider.

OLE:

These connection strings vary according to the OLE database provider.

Once the database type is set and the database connection is opened, you may use several more commands.

```
SQL_EXECUTE [SQL or Text File]
SQL_EXEC_TO_VAR [variable name] | [SQL]
DATAKEY_TO_SQLTABLE [table name]
DATAKEY_FROM_SQLTABLE [table name]
```

The SQL_EXECUTE command will execute an arbitrary SQL command, which combined with variables can be quite useful. Alternatively, it can be fed the name of a text file and the contents of the text file will be passed and executed as a series of SQL commands.

The SQL_EXEC_TO_VAR takes a variable ID and a SQL statement and passes the results of the SQL command to the given variable. In this way, data may flow from a database system to the JobScript.

(See a previous section of this chapter, SQL Tables, for information on DATAKEY commands.)

File Commands

```
LOG_WRITE [message] | [file name]
COPY [source file name] | [destination file name]
```

DELETE [file name]
MOVE [file name] | [destination file name]
PATH [variable name] | [variable name]
FILENAME [variable name] | [variable name]

These commands help scripts manipulate files, including logging, moving files to different places, and renaming them.

The LOG_WRITE command will append a message to a given text file. Using variables, it is possible to log a rich set of circumstances about the operation, including errors encountered during the encoding/decoding process ($$ERRLOG$$), critical errors, ($$ERR_CRIT$$), the number of rows in the Data Key ($$DK_ROWS), and so on.

The COPY command copies the file given in the source to the destination. A filename must be supplied for the destination.

The DELETE command removes the target file from the file system.

The MOVE command moves a file from a source to a destination. It is important to note that it is expecting a full file name for the destination as well as the source.

PATH extracts just the path portion from a full file path in the first specified variable and stores it in the second variable.

FILENAME extracts just the filename portion from a full file path in the first variable and stores it in the second variable.

Example Script:

```
@FOR 1|DIR|$$PATH$$\INCOMING
      COPY $$1$$|$$PATH$$\PROCESS\CURRENT_HIPAA.TXT
      ...
      FILENAME 1|3
      @IF $$ERR_CRIT$$|=|0
            MOVE $$1$$|$$PATH$$\COMPLETED\$$3$$
            LOG_WRITE $$1$$ PROCESSED|$$PATH$$\LOGS\837.LOG
      @ELSE
            MOVE $$1$$|$$PATH$$\ERRORED\$$3$$
            LOG_WRITE $$1$$ CRITICAL ERROR|$$PATH$$\LOGS\837.LOG
      @ENDIF
@ENDFOR
```

In this example, the above commands are tied together to form a script fragment. The @FOR/@ENDFOR processes all files within the INCOMING directory off the base Chiapas install path. The '...' indicates some form of processing, probably decoding. Then, only the filename is extracted from the full path stored in Variable 1 and saved to Variable 3.

After this, an @IF conditional branch instruction checks to see if there was a critical error. The '0' indicates that there were no critical errors; since there were no critical errors, the MOVE command moves the original file to the COMPLETED directory and the LOG_WRITE command logs it as being successfully processed.

If there was a critical error, the file is placed in the ERRORED directory and the log updated to

reflect this.

External Execution Commands

Often, the functionality within the JobScript is not enough to carry out specific tasks. The language supports external task execution via four commands used for passing control externally for a short while.

```
WAIT_FOR_BATCH [file name]
EXECUTE_BATCH [file name]
PARSE_FILE_NEWSTATE [file name] | [argument 1] | [argument 2]
PARSE_FILE_OLDSTATE [file name]
```

The first command, WAIT_FOR_BATCH, passes execution to the operating system and executes the specified file name. It will wait for execution to complete before continuing. The second command, EXECUTE_BATCH, will let the command execute on its own and not wait for it to complete.

PARSE_FILE_NEWSTATE calls another JobScript as a subroutine. It will create a new, blank Job State and pass two arguments to it, which become variables 1 and 2 in the new Job State. When the script completes, variables 99 and 100 in the calling JobScript will be set from variables 99 and 100 from the called JobScript. In this way, any results you want to return to the calling script can be returned via variables 99 and 100.

HIPAA Transport Operations

Chiapas presents several commands for saving and loading HIPAA files, as well as manipulating their format. HIPAA files can vary according to the following:

```
HIPAA_FROM_FILE [file name]
HIPAA_TO_FILE [file name]
SET_SPECIAL [special name] | [value]
```

The first two commands relate to Chiapas' interface with HIPAA files. Chiapas can natively determine several attributes of a file that may not be strictly HIPAA compliant but are found to be used "in the field," including 80-column formatting and end-of-segment Carriage Return/Line Feeds.

The HIPAA_FROM_FILE command loads a HIPAA file into the internal SegPool structure, and sets the internal variables relating to how it is formatted, such as segment terminators and element separators. The HIPAA_TO_FILE writes the SegPool structure to disk. A HIPAA_FROM_FILE followed by a HIPAA_TO_FILE to another filename should result in identical files. It may be necessary to change the formatting of the HIPAA file, so this is where the SET_SPECIAL command comes into play.

The SET_SPECIAL command allows you to directly set several internal variables. They are:

ICNSEED – Sets the seed for the ICN. This number should be nine digits long.
SP_80COL – If the new value is one, sets the 80 Column Boolean to true for the HIPAA object.
SP_CHRELE – Sets the HIPAA object element separator character. This is an ASCII value for the character, not the actual character itself.

SP_SEGTRM – Sets the HIPAA object segment terminator character (the same way as it does for SP_CHRELE).
SP_SUBTRM – Sets the HIPAA object sub-element separator character, again as above.
SP_CR – Sets the HIPAA object carriage-return flag ('1' or '0').
SP_LF - Sets the HIPAA object line-feed flag ('1' or '0').

If you wanted to make a HIPAA file easier to read, this code snippet would be helpful:

```
HIPAA_FROM_FILE C:\EDI\SOURCE_HIPAA.TXT
SET_SPECIAL SP_CR|1
SET_SPECIAL SP_LF|1
SET_SPECIAL SP_SEGTRM|126
HIPAA_TO_FILE C:\EDI\DEST_HIPAA.TXT
```

In this short example, a HIPAA file is loaded into the internal SegPool object, the linefeed and carriage return flags are set, followed by setting the Segment Terminator to 126 (the '~' character in ASCII). The element separator character (SP_CHRELE) cannot be set this way; it must be set prior to an ENCODE operation. Once the internal variables are set, the HIPAA_TO_FILE operation saves the resulting HIPAA file to disk.

Saving variables & Setting Counters

```
SET [variable name] | [value]
SET_FROM_FILE [variable name] | [file name]
SET_TO_FILE [variable name] | [file name]
SUM [variable name] | [number]
```

JobScript provides for saving variables to and from disk. The SET_FROM_FILE loads a variable from a text file on disk; SET_TO_FILE saves the variable to a text file on disk. This will overwrite it if it already exists. The SET command stores the value straight to the variable.

The SUM command works for when the variable contains a number. It will add a certain amount, positive or negative, to that variable and update it to the result. If Variable 1 contained '567' and SUM 1|3 was sent, then Variable 1 would contain '570' afterwards.

This can be useful for Interchange Control Numbers (ICN). ICNs define a unique identifier for each file, and need to increase every time they are used. In this example JobScript code, a text file containing a 9-digit ICN number called 'MCR_OUT' is read from, increased by three, and saved, as well as setting the ICNSEED variable. It is presumed the ENCODE operation will take place soon after.

Example:

```
SET_FROM_FILE 2|C:\EDI\VERSION1\COUNTERS\MCR_OUT.TXT
SET_SPECIAL ICNSEED|$$2$$
SUM 2|3
SET_TO_FILE 2|C:\EDI\VERSION1\COUNTERS\MCR_OUT.TXT
```

Encode/Decode Operations

```
TREE_FROM_FILE [file name]
MAPKEY_FROM_FILE [file name]
```

ENCODE
DECODE

Up to this point, we have covered looping and conditional constructs, file commands, database commands, and external execution commands. Now, we will address the heart of the JobScript: encoding and decoding.

Any encoding and decoding operation has three prerequisites. Both require a loaded Hierarchy Tree and Map Key. Encoding requires a loaded Data Key, and Decoding requires a loaded SegPool object. To load the Hierarchy Tree, use the TREE_FROM_FILE command. Likewise, the Map Key is loaded via the MAPKEY_FROM_FILE. Both the Hierarchy Tree and Map Key files can be maintained via the chiapasManager application.

Once all three prerequisites are fulfilled, then the ENCODE or DECODE call is made.

An ENCODE operation will populate a SegPool object from the loaded Data Key, and any errors will be added to the internal Error Log (accessed via the $$ERRLOG$$ variable).

A DECODE operation does creates a Data Key from the loaded SegPool, and it also alters the SegPool to reflect the Loop Index. It is no longer a valid SegPool once it is used, but the information it contains can be useful when building extended mapping information for Map Keys for specific business situations. Also, the DECODE operation will log all detected HIPAA syntax errors to the error log, and it will populate an Ack Key that can be used to create a Functional Acknowledgment.

Functional Acknowledgments

DATAKEY_FROM_ACK

After a successful DECODE operation, a special Data Key is stored in the Ack Key variable. This contains a log of decoded envelopes from the incoming HIPAA file. If there was a critical error, it will specify which segment caused the failure. This key goes hand-in-hand with the MAP_997.KEY Map Key found within the default Chiapas installation, and can be used to create a Functional Acknowledgment file that can be given to the sending trading partner as proof of receipt of their information. This script example shows how to save the resulting file to a filename given in Variable 50:

```
' Set the Data Key, encode, save to 997.
DATAKEY_FROM_ACK
MAPKEY_FROM_FILE $$PATH$$\MAPKEYS\MAP_997.KEY
TREE_FROM_FILE $$PATH$$\TREE\TREE_EXP.BIN
SET_SPECIAL SP_LF|0
SET_SPECIAL SP_CR|0
ENCODE
HIPAA_TO_FILE $$50$$
```

Handling Errors

There are three variables set aside for errors:

$$ERR_COUNT$$
$$ERR_CRIT$$
$$ERRLOG$$

There are three variables set aside for errors. $$ERR_COUNT$$ yields the number of errors within the Error Log; $$ERR_CRIT$$ is a 0 or 1 depending on if there was a critical encoding or decoding error that stopped the process; and $$ERRLOG$$ is a complete listing of all of the errors.

This short example writes all of the errors to an error log called “MCR.LOG.”

```
LOG_WRITE $$ERRLOG$$|C:\EDI\VERSION1\LOGS\MCR.LOG
```

Appendix A: JobScript Reference

Global Variables

$$[1 - 100]$$

This is a reference to one of 100 global variables in JobScript. They are numbered 1 through 100. If the variable is specified directly in an argument, just the number is used; if it is part of an expression, then it is surrounded by two dollar signs, where it will then evaluate out to the contents of the variable.

Example:

```
SET 1|Hello.
ECHO $$1$$
```

Results in:

```
Variable 1 set to Hello..
Hello.
```

$$MK_ROWS$$

If the MapKey object is set, this will return the number of mappings inside. If the MapKey object is not set, this evaluates to a zero.

$$DK_ROWS$$

If the DataKey object is set, this returns the number of rows in the Data Key. Otherwise, a zero is returned.

$$AK_ROWS$$

Whenever a HIPAA file is decoded, an Acknowledgement Data Key is generated that allows for the generation of 997 response files. If this is set, it returns the number of rows in the Ack Key; otherwise, zero is returned.

$$SP_SEGS$$

If a HIPAA file is successfully loaded, this returns the number of segments therein. Otherwise, a zero is returned.

$$ERR_COUNTS$$

Whenever an encode or decode operation is carried out, any errors that come up during the process are stored in the Error Log object. This returns the count of errors.

$$ERR_CRIT$$

If a critical error occurs during an encode or decode operation, this returns a '1.' Otherwise, '0' is returned. Currently, only one error triggers an ERR_CRIT to be 1, and that is ERROR_H013 – unknown segment, which stops the file from being decoded.

$$ERRLOG$$

This returns all of the errors inside of the Error Log object in a number of consecutive lines.

$$DT8$$

Returns an 8-digit date, in the format of YYYYMMDD.

Example:

```
20040812
```

$$DATE$$

Returns a western format date string, in the format of MM/DD/YYYY.

Example:

```
08/12/2004
```

$$TM4$$

Returns a 24-hour four digit timestamp, in the format of HHMI.

Example:

```
1308
```

$$TIME$$

Returns a western style time string, in the format of HH:MI AM/PM.

Example:

```
1:09 PM
```

$$SP_80COL$$

If a HIPAA file is loaded, this evaluates to a '1' or '0' depending on if the HIPAA file is in 80-column format.

$$SP_CHRELE$$

If a HIPAA file is loaded, this evaluates to the element separator character.

$$SP_SEGTRM$$

If a HIPAA file is loaded, this evaluates to the segment separator character.

$$SP_SUBTRM$$

If a HIPAA file is loaded, this evaluates to the sub-element separator character.

$$SP_CR$$
If a HIPAA file is loaded, this evaluates to a '1' or '0' depending on if the HIPAA file is line separated with CR (Carriage Return) characters.

$$SP_LF$$

If a HIPAA file is loaded, this evaluates to a '1' or '0' depending on if the HIPAA file is line separated with LF (Line Feed) characters.

Flow Commands

@IF [expression 1] | [= OR < OR > OR <> OR <= OR >=] | [expression 2]
…
@ELSE
…
@ENDIF

The @IF command establishes a conditional branch depending on the comparison of expression 1 and expression 2. If the Boolean expression is true, the section from @IF…@ELSE (or @IF… @ENDIF if @ELSE is not present) is evaluated. If the @ELSE expression is present, the @ELSE…@ENDIF is evaluated instead.

@FOR [variable name] | NUM | [start number], [end number], [increment]
…
@ENDFOR

Establishes a loop that iterates through the set of number values given the increment. For each instance, the current number is placed into the given variable name. The sign must be of a value that does not result in an infinite loop.

Example 1:

```
@FOR 1|NUM|1,5,1
ECHO $$1$$
@ENDFOR
```

Results in:

```
1
2
3
4
5
```

Example 2:

```
@FOR 1|NUM|1,5,-1
ECHO $$1$$
@ENDFOR
```

Results in:

```
Error: @FOR increment is an invalid sign which will result in an
infinite loop.
```

@FOR [variable name] | DIR | [directory name]
…
@ENDFOR

This iterates the @FOR…@ENDFOR loop for each instance of a file. If no files are present in the directory, the loop is skipped. If there is no directory at all by that name, an error message is generated.

Example:

```
@FOR 1|DIR|C:\edi\version1\scripts_pending
ECHO $$1$$
@ENDFOR
```

(This will output all the filenames within the given directory.)

@FOR [variable name] | SQL | [SQL statement]
…
@ENDFOR

Sometimes it is necessary to scan a SQL table and handle a number of items given therein. This command allows executing one SQL statement where the first column of each row contains a VARCHAR/string. The loop is then repeated for each row returned, with the result going into the specified variable name. As shown, this depends on both the DBTYPE being set and an open SQL connection.

Example 1:

```
DBTYPE SQLSERVER
CNN_OPEN Data
Source=(local);chiapas_edi_production;Trusted_Connection=true;
@FOR 1|SQL|SELECT DISTINCT CONVERT(VARCHAR,CLAIMNO3) FROM RESULTS
ECHO $$1$$
@ENDFOR
```

Results in:

```
847217
991645
582984
297573
…
```

Example 2:

```
@FOR 1|SQL|SELECT DISTINCT CONVERT(VARCHAR,CLAIMNO3) FROM RESULTS
ECHO $$1$$
@ENDFOR
```

Results in:
```
Error: Error handling @FOR initialization.  The @FOR statement requires
an open SQL Connection object.
```

@FOR [variable name] | TXT | [file name]
…
@ENDFOR

This yields a loop through the lines of a given text file.

Example 1:

```
@FOR 1|TXT|C:\edi\version1\
ECHO $$1$$
@ENDFOR
```

Results in:
```
Error: Error handling @FOR initialization.  File does not exist.
```

Example 2:
```
@FOR 1|TXT\C:\edi\version1\scripts_helper\test_hello.txt
ECHO $$1$$
@ENDFOR
```

Results in:
```
ECHO HELLO
```

@FOR [variable name] | ERR
…
@ENDFOR

This command establishes a loop for every line in the Error Log object. With this, one can check for the existence of a certain type of error or put the error records independently into a text-based error log.

General Instructions

HELP [HELP OR Instruction]

HELP HELP will pull up a list of all instructions. HELP [instruction name] will pull up the documentation for that particular instruction.

LOAD_ERRLOG_LINE [variable name] | [line number]

Loads the specified entry in the error log into the given variable name.

DBTYPE [database type]
Database type is within (SQLSERVER, SQLITE, ORACLE, ODBC, OLE). This command must be executed before a database connection can be opened to set the type of database to connect to.

CNN_OPEN [connection string]

Opens a database connection to the specified .NET connection string. This command needs to be executed before any SQL statements can be parsed.

CNN_CLOSE

Closes the open connection.

LOG_WRITE [message] | [file name]

This writes a message out to a given file name. If the log does not exist, it is created.

Example:

```
LOG_WRITE $$DK_ROWS$$ Rows were created for filename $$1$$ |
c:\edi\version1\logs\pro_log.log
```

DATAKEY_FROM_UB92 [UB92 file name] | [ReMap file name] | [UB92 Definitions file name]

This will extract a Data Key from a UB92 data file. The format for the UB92 format itself is present in the UB92 Definitions filename. The listing of the data to be extracted from the file is listed in the ReMap file name. The ReMap filename is formatted as follows:
Example:

```
REND_LICENSENO
BILLPROV_NAME           10    11
BILLPROV_EMPID          10    3
BILLPROV_ADD1           10    12
REND_PROVEIN
REND_LICENSENO          80    4
```

For this simple definition, the first line represents the “new row” condition that creates a new row in the Data Key. The REND_PROVEIN is a blank column, and the other four contain UB92 specific row ID/column ID pairs the reference a specific line in the UB92 Definitions file. From this information, a UB92 data file can be processed and usable business information can thereby be extracted.

The UB92 Definitions file is part of the standard deployment for Chiapas.

Example:

```
DATAKEY_FROM_UB92 C:\EDI\VERSION1\TRANSPORT_IMPORT\UB92_IMPORT\UB92.TXT|
C:\EDI\VERSION1\BIN\TABLE_DEF.TXT|C:\EDI\VERSION1\BIN\UB92DATA.TXT
```

DATAKEY_FROM_SQLTABLE [table name]

Given an open connection, this will extract a Data Key from a table of the given name.

DATAKEY_FROM_ACK

Transfers the Acknowledgement Key into the active Data Key object. Given a proper Mapping Key and then an ENCODE call, a 997 response file can be created.

DATAKEY_TO_CSV_NOHEADER [CSV file name]
DATAKEY_TO_CSV_HEADER [CSV file name]

Transfers the active Data Key object into a Comma-Separated Values file. The two commands are used depending on whether a header row is desired.

DATAKEY_FROM_CSV_NOHEADER [CSV file name]
DATAKEY_FROM_CSV_HEADER [CSV file name]

Extracts a Data Key from a Comma-Separated Values file. The two commands are used depending on whether there is a header row present in the file.

DATAKEY_TO_SQLTABLE [table name]

Given an active SQL connection, this transfers the active Data Key into a SQL table of the given name.

MAPKEY_FROM_FILE [file name]

Deserializes a MapKey object from the given filename. The filename must be a valid serialized MapKey object.

MAPKEY_TO_FILE [file name]

Serializes the MapKey object to the given filename. Normally used for Legacy operations.

HIPAA_FROM_FILE [file name]

Loads a HIPAA file. The various flags for 80 columns, line feed, carriage return, etc. and the separator characters can be read via the global variable values (such as $$SP_CHRELE$$ and so on).

HIPAA_TO_FILE [file name]

Saves a HIPAA file to disk. The file will be formatted according to the set global values (such as $$SP_CHRELE$$ and so on).

DECODE

Given loaded HIPAA file, Tree and MapKey objects, this command actually executes the decoding pass. Any errors resulting from the decode process will be placed in the Error Log object. The resulting information will be placed in the Data Key object. Furthermore, the HIPAA object is destructively modified to reflect loop-numbering information and cannot be used again for decoding. A HIPAA_TO_CSV command will show how the loop information was processed by

the Chiapas Decoder; this information can be used for absolute referencing in mappings.

After decode, the Data Key object can be saved to a CSV file via DATAKEY_TO_CSV_HEADER [file name] or to a SQL table via DATAKEY_TO_SQLTABE [table name] (open connection required).

ENCODE

Given loaded Data Key, Tree and MapKey objects, this command will execute the Chiapas Encoder and create a new HIPAA object. Any errors resulting from the encoding process will be placed in the Error Log object. Afterwards, the HIPAA object global values can be set via the SET_SPECIAL command and then saved to disk via the HIPAA_TO_FILE command.

TREE_FROM_FILE [file name]

Deserializes the Tree object from a binary file on disk. This is the hierarchy data representing all HIPAA specifications, and this data may be edited via the ChiapasManager application. A new Tree should be deserialized before every encode or decode operation.

TREE_TO_FILE [file name]

Saves the Tree to a file. This is normally used in conjunction with LEGACY_NEWTREE, as JobScript does not support active manipulation of the Hierarchy Tree.

LEGACY_NEWTREE [connection string]

Given a legacy Chiapas RC1 database tree, this loads in a Version 1 tree structure by parsing the old data. The tree can then be saved to disk via the TREE_TO_FILE command.

LEGACY_MAPKEY_FROM_RC1 [BizMap Header ID] | [connection string]

Given a BizMap Header ID of a Chiapas RC1 mapping table and a connection string, this will parse the data into a Version 1 MapKey object. This object can then be saved to disk via the MAPKEY_TO_FILE and then edited with the ChiapasManager application.

SET [variable name] | [expression]

Sets the given variable to the expression.

Example:

```
SET 1|$$TIME$$
ECHO $$1$$
```

Results in:

```
2:33 PM
```

LOAD_ELEMENT [variable name] | [segment #] | [element #]

Loads an element from the HIPAA object given at the indicated segment and element and stores it in the specified variable. Useful for referencing Interchange Control Numbers from a HIPAA file

for storing inside of job tables and so on.

LOAD_SEGMENT [variable name] | [segment #]

Loads a specified segment line from the given HIPAA object and stores it in the given variable.

LOAD_DATAKEY_CELL [variable name] | [row #] | [column #]

Loads the given Data Key cell inside the loaded Data Key and stores the referenced cell (given by row and column #s) and stores it inside the given variable.

SQL_EXECUTE [SQL or Text File]

If the argument resolves to a valid text file, then the text file is treated as a list of SQL commands and they will be read and executed. If it does not, then it will be treated as normal SQL instructions and executed. An active SQL connection is required.

Example 1:

```
SQL_EXECUTE TRUNCATE TABLE DATAKEY_834
```

```
SQL_EXEC_TO_VAR [variable name] | [SQL]
```

Emits the first column of the first row from the results of the given SQL statement and stores it in the variable.

Example 2:

```
SQL_EXEC_TO_VAR 1|SELECT COUNT(*) FROM DATAKEY_834
```

This will store the count of rows inside of the SQL table DATAKEY_834 and place them in Variable 1.

ECHO [expression]

Echoes the expression back to the results. If ECHO CLEAR is given, the current results string is cleared.

WAIT_FOR_BATCH [file name]

Executes a DOS BATCH file and waits for execution to complete.

EXECUTE_BATCH [file name]

Executes a DOS BATCH file and continues; it does not wait for execution to complete.

PARSE_FILE_NEWSTATE [file name]

Calls the JobScript parser on the given script, creating a new context (new variables and so on). Variables 99 and 100 are transferred from the new context back to the old context after execution is complete, allowing for some limited information to travel back to the calling script; however the Data Key, HIPAA, Map Key and Tree objects in the new context are all destroyed after the

external script completes execution.

PARSE_FILE_OLDSTATE [file name]

Calls the JobScript parser on the given script, reusing the same context for the new execution. All variables and objects will carry over to the new script, and these variables and objects can be changed by it as well.

MOVE [file name] | [destination]

Moves the given file name to the new destination. The destination must include a complete file name, not just a path.

PATH [variable name] | [variable name]

Given a full path and filename in the first variable, the path will be extracted and stored in the second variable.

FILENAME [variable name] | [variable name]

Given a full path and file name stored in the first variable, just the filename (with extension) will be extracted and stored in the second variable.

MAPKEY_ADD [key name] | [key value]

Adds a mapping with the given key name and value to the mappings of the current Map Key. This can be useful for adding new mappings depending on the situation.

CUT [variable name] | [begin character] | [length]

Replaces the contents of the given variable name with a substring of itself, beginning at the specified beginning character for the length specified in the third argument.

REPLACE [variable name] | [target string] | [replacement string]

Replaces the contents of the given variable by replacing the target string with the replacement string.

SUM [variable name] | [integer]

Adds the given integer (positive or negative values allowed) to the value stored in the given variable name. If the variable name does not contain a number value, an 'Error: Summation Failed' error is raised.

HIPAA_TO_CSV [file name]

Stores the loaded HIPAA file to a comma separated values file (CSV). If the HIPAA file has been decoded, the segments will be preceded by 10 comma-separated numbers indicating the loop-level values for each segment. If the DECODE command has not been processed for that object, it will just reflect sequential segment and element values.

SET_DATAKEY_CELL [variable name] | [row number] | [column number]

Sets the given Data Key cell (referenced by column and row) to the value stored in the given variable.

SET_FROM_FILE [variable name] | [file name]

Loads a text file into the given variable.

SET_TO_FILE [variable name] | [file name]

Saves the specified variable to a text file at the given filename.

SET_SPECIAL [special] | [new value]

This allows the script to set new values for certain global values, including the Interchange Control Number seed and all of the general values for HIPAA file formatting.

They include:
ICNSEED – Sets the seed for the ICN. This number should be nine digits long.
SP_80COL – If the new value is '1' it sets the 80 Column Boolean to 'true' for the HIPAA object.
SP_CHRELE – Sets the HIPAA object element separator character.
SP_SEGTRM – Sets the HIPAA object segment terminator character.
SP_SUBTRM – Sets the HIPAA object sub-element separator character.
SP_CR – Sets the HIPAA object carriage-return flag ('1' or '0').
SP_LF - Sets the HIPAA object line-feed flag ('1' or '0').

PARSE

One limitation of the JobScript parser is that during Shell mode, SQL connection objects do not stay open between parsed job lines. Also, @FOR…@ENDFOR loops are not normally possible in this context. As a way to get around these limitations, this command allows you to execute a series of commands sequentially in separate context. In this context, connection operations and @FOR…@ENDFOR loops are possible.

To do this, different special characters are used within this new context. For example, $$1$$ will resolve to variable 1 BEFORE the expression is evaluated, making it impossible to refer to the variable within the context. Specifically:

^ - line separator (used to separate each line)
# - variable character (used in all places where the $ is used)

Example:

```
PARSE @FOR 1|DIR|c:\edi\version1\scripts_complete^ECHO ##1##^@ENDFOR
```

Results in:
… directory contents of c:\edi\version1\scripts_complete…

DATAKEY_FROM_FLAT [format file name] | [flat file name]

Returns a Data Key from a fixed-length flat text file. Since the columns must contain a certain

formatting, the formatting must be defined first via the given format file name. A format file is a CSV file containing two fields: field length and field name.
Example:

```
3,VERSION
20,LNAME
15,FNAME
```

This example would define a flat file 38 characters wide with three fields.

Given a valid format file and a flat file name following this format, a Data Key is loaded from it.

DATAKEY_TO_FLAT [format file name] | [flat file name]

This is the converse of the above command, DATAKEY_FROM_FLAT. Given a valid format file as described in the DATAKEY_FROM_FLAT command and a destination path and name for the flat file, a new flat file will be created from the loaded Data Key.

DELETE [file name]

Deletes the specified file.

COPY [source file name] | [destination path]

Copies the specified file to the destination path.

BEGIN_HIPAA
…
END_HIPAA

Begins copying of an inline HIPAA file from console; every line must contain a new segment. END_HIPAA will load the captured file to the HIPAA object

BEGIN_CSV
…
END_CSV

Begins copying an inline CSV file from console. It must contain a header. END_CSV will load the captured CSV file to the Data Key.

ECHO_CSV

Echos the loaded Data Key to the console. ‘CSV_START’ will be echoed, then the CSV file with header line, and then a ‘CSV_END’ line.

ECHO_HIPAA

Echos the loaded HIPAA object. ‘HIPAA_START’ will be echoed, then the HIPAA file with one line per segment, and then a ‘HIPAA_END’ line.

Appendix B: Error Messages

Error List:

S - JobScript Module errors
E - Encoder-specific errors
H - HIPAA syntax errors
X - General resource errors
C – Database-specific errors

Many errors will be followed by an additional line that gives specific information about the error message.

INTERFACE MODULE

General Resource Errors:

ERROR_X001: There was difficulty with the passed X12SegPool object.

ERROR_X002: There was difficulty file referencing the HIPAA file.
This occurs when the HIPAA parser encountered difficulty splitting the incoming HIPAA file into segments and elements. Generally, this occurs when the referenced file is not encoded to HIPAA specifications, or is not an actual HIPAA file at all.

ERROR_X003: General Failure with loading HIPAA file / object.
This is triggered by a general failure to decode the HIPAA file.

ERROR_X004: There was difficulty object referencing the tree.

ERROR_X005: There was difficulty loading the tree from the filename <filename>.
This fires when Chiapas is unable to load a valid Hierarchy Tree from the disk. Verify that the path is correct to the Chiapas tree file.

ERROR_X006: General Failure with loading the tree object.
This fires for a general failure to load a valid Hierarchy Tree.

ERROR_X007: There was difficulty object referencing the mapping key.
This error is triggered by a corrupted MapKey file.

ERROR_X008: There was difficulty loading the mapping key from the filename <filename>.
This is fired when Chiapas has difficulty accessing a MapKey file.

ERROR_X009: General Failure with loading the mapping key object.
A general catchall message when a MapKey fails to load.

Transport Errors:

ERROR_C001: There was difficulty referencing the X12SqlConnection object.

ERROR_C002: There was difficulty using the connection string.
This fires whenever a connection cannot be established given the connection string and given DBType. The database type needs to be set via the DBTYPE command, and the connection string must be to a valid database server and database.

ERROR_C003: There was difficulty opening the database connection.
This is fired by a database transport layer when the given connection is rejected.

ERROR_C004: There was difficulty creating the SQL table on the supplied connection.
A DecodeToSQLTable command failed to emit the decoded information to a table on the given connection.

JOBHANDLER MODULE

ERROR_S001: The third @FOR argument requires three numbers (ex: 1,5,1 or 5,1,-1)
The numeric @FOR statement requires three numbers: start number, end number, and the increment.

ERROR_S002: @FOR increment is an invalid sign which will result in an infinite loop.
The given increment will turn the @FOR loop into an infinite loop.

ERROR_S003: Directory does not exist.
This is triggered by a @FOR statement referencing a non-existent directory.

ERROR_S004: The @FOR statement requires an open SQL Connection object.
When a @FOR statement is made with a SQL statement, there must be an active, open connection made with the CNN_OPEN command to work.

ERROR_S005: There was a problem evaluating the SQL expression
This message will fire when the SQL expression inside a @FOR statement fails to resolve properly.

ERROR_S006: File does not exist.
Triggered by a @FOR statement trying to reference a text file that cannot be found. This can also be triggered by other statements seeking a valid file name.

ERROR_S007: There was difficulty evaluating the directory
This is a general catchall error for failure to read the text file referenced in a @FOR statement.

ERROR_S008: Error handling @FOR initialization.
General Failure for setting up a @FOR statement. The parser will try to resume execution after the @ENDFOR.

ERROR_S009: General Failure reading JobScript file
If Chiapas is called to execute a script that is inaccessible, this error triggers.

ERROR_S010: DBTYPE not set.
Database connections cannot be opened until the DBTYPE is set.

ERROR_S011: Acknowledgement Key not set.

The Ack Key will not be set until a successful DECODE operation, and can not be used until then.

ERROR_S012: Connection not opened.
This is triggered when a command depends on a SQL connection that is not opened.

ERROR_S013: DataKey is not yet loaded to carry out this operation.
This fires when the Data Key is referenced by a command when it is not set or populated.

ERROR_S014: MapKey not set.
This fires when the Map Key is referenced by a command, and it has not been loaded.

ERROR_S015: HIPAA file not loaded.
The Internal SegPool object, which represents a valid HIPAA file, is being referenced by a command when the SegPool is empty or unset.

ERROR_S016: Tree not set.
This fires when the Hierarchy Tree is not set, but is being referenced by a command.

ERROR_S017: @IF failure.
When the parser has difficulty with the arguments in an @IF command, this error is triggered.

ERROR_S018: @ELSE failure.
If the @ELSE appears without a corresponding @IF, or multiple @ELSE clauses appear for one @IF, this error will trigger.

ERROR_S019: @ENDIF failure.
If an @ENDIF appears without a mated @IF clause or some other failure occurs in parsing, this error will fire.

ERROR_S020: @FOR stack empty.
This error can fire when a @ENDFOR is presented without a matching @FOR statement.

ERROR_S021: @ENDFOR Failure.
This is a general failure to resolve the next item in a @FOR/@ENDFOR loop.

ERROR_S022: Syntax error in line
This error occurs when a '@' prefixed command is unrecognized by the parser. This can also occur if a command was expecting an argument that was left empty.

ERROR_S023: Invalid Variable
This occurs when Chiapas fails to parse a variable name surrounded by double dollar signs.

ERROR_S024: This is not a valid integer.
This fires when a JobScript argument cannot be converted to an integer for a command that is expecting one.

ERROR_S025: A valid variable name is a number between 1 and 100.
Any arguments expecting a variable are restricted to the valid range of 1 through 100 for variable names.

ERROR_S026: Valid file name required.
This triggers when an invalid path or invalid filename is passed as an argument to a JobScript command.

ERROR_S027: Error line out of range.
This failure results from trying to access an ERRLOG line that is out of range.

ERROR_S028: An invalid DB type was passed. Use HELP DBTYPE for valid arguments for this command.
An invalid database type was provided to the DBTYPE command. The only valid database types are SQLSERVER, SQLITE, ODBC, OLE, and ORACLE.

ERROR_S029: There was difficulty opening the connection.
The connection string was rejected by the data provider specified in DBTYPE, so the connection is not active.

ERROR_S030: There was difficulty closing the connection.
Chiapas had trouble closing an open database connection.

ERROR_S031: Log Write Failure.
Chiapas failed to add to the specified log filename. The filename may be invalid or not available for writing.

ERROR_S032: The UB92 failed to decode.
There was general failure decoding a UB92 data file. The file may be invalid, or the initialization files may be incorrect.

ERROR_S033: DataKeyFromTable failure
Chiapas had difficulty extracting a Data Key from a SQL table.

ERROR_S034: There was difficulty writing the CSV file.
This is a general failure for writing to a Comma-Separated Values file. The file may already exist or the path may be incorrect.

ERROR_S035: There was difficulty parsing the CSV file.
This results when Chiapas is unable to create a Data Key from a Comma-Separated Values file. The CSV file may be corrupt.

ERROR_S036: There was difficulty accessing the file.
The CSV file may be locked or not available for unable to be read.

ERROR_S037: DataKeyToTable failure
This error results when Chiapas fails to save a Data Key to a SQL table on an active SQL connection.

ERROR_S038: MapKey failed to load.
The specified file does not resolve to a valid Map Key file.

ERROR_S039: The filename supplied is invalid.
Chiapas failed to save the Map Key to the given filename.

ERROR_S040: HIPAA file failed to load.
The specified file can not be resolved as a valid HIPAA file. The initial ISA segment may be an invalid size, or it may not be a HIPAA file at all.

ERROR_S041: Encode failure - see error log for details.

Chiapas failed to encode a new SegPool object. Exactly why will be given in the internal Error Log, accessed via the $$ERRLOG$$ variable.

ERROR_S042: There was difficulty opening the supplied connection string.
The legacy RC1 Chiapas database can not be opened with the specified connection.

ERROR_S043: Invalid Element Number.
The LOAD_ELEMENT command failed because the given element in the SegPool object was invalid.

ERROR_S044: Invalid Segment Number.
The LOAD_ELEMENT command failed because the given segment in the SegPool object was invalid.

ERROR_S045: Invalid Segment Number.
A LOAD_SEGMENT command failed because the segment inside the SegPool object was invalid.

ERROR_S046: Invalid Column Number.
A LOAD_DATAKEY_CELL command failed because the referenced column in the Data Key does not exist.

ERROR_S047: Invalid Row Number.
A LOAD_DATAKEY_CELL command failed because the referenced row in the Data Key does not exist.

ERROR_S048: An error happened while processing the SQL statement.
The SQL passed to the SQL_EXECUTE was rejected by the database provider.

ERROR_S049: An error happened while processing the SQL statement.
The SQL was rejected by the database provider on the SQL_EXEC_TO_VAR command, and no variable was set.

ERROR_S050: Batch Execution problem
There was difficulty creating a new process on the WAIT_FOR_BATCH command.

ERROR_S051: Batch Execution problem
There was difficulty creating a new process for the EXECUTE_BRANCH command.

ERROR_S052: An error occurred while trying to move the file.
This indicates Chiapas had trouble moving a file. It is important to note that the destination argument in the FILE_MOVE command requires both the path and a filename for this command to function.

ERROR_S053: An error occurred while extracting the path.
There was an error extracting just the filename from the provided string in the PATH statement. It may not be a valid path or filename.

ERROR_S054: An error occurred while extracting the filename.
Chiapas had trouble extracting the filename from the given string. There may be trouble with the path/filename syntax.

ERROR_S055: Cut failed.
The CUT command failed; there may have not been enough characters in the string.

ERROR_S056: Replace failed.
The REPLACE command failed.

ERROR_S057: Summation failed.
The SUM command failed. Some of the supplied arguments may contain non-numeric strings.

ERROR_S058: Write to CSV failed.
Chiapas failed to write the Data Key object to a Comma-Separated Values file. The file may already exist.

ERROR_S059: Invalid Column Number.
The SET_DATAKEY_CELL failed because the specified column number is out of range.

ERROR_S060: Invalid Row Number.
The SET_DATAKEY_CELL failed because the specified row number is out of range.

ERROR_S061: File error
There was a failure reading the indicated file given to the SET_FROM_FILE command.

ERROR_S062: Must provide a valid numeric expression for argument 2.
The SET_SPECIAL command requires that the second argument is a valid number.

ERROR_S063: SegPool object not set.
SegPool-related variables cannot be set until the SegPool object is created.

ERROR_S064: Error converting number to character.
The number must be within the range of 0 and 255 to convert to a correct separator character.

ERROR_S072: DataKey error.
This is a general failure message that occurs during a DATAKEY_FROM_FLAT command. The flat file CSV specification must exist, and must match the given CSV file for this command to work.

ERROR_S073: DataKey error.
This error is the same as ERROR_S073, but occurs during a DATAKEY_TO_FLAT command.

ERROR_S074: An error occured while trying to copy the file.
Chiapas failed to copy the source file to the destination.

ERROR_S075: ?SYNTAX ERROR
This error occurs on any statement that is not a valid JobScript statement.

ERROR_S076: There was trouble accessing the indicated SQL file.
If a SQL_EXECUTE statement is given an incorrect .SQL text file, this error will trigger.

ERROR_S077: There was difficulty saving the HIPAA file to disk.
This error triggers when the HIPAA_TO_FILE command fails to write a new HIPAA file to disk.

TRANSPORTSQL MODULE

These errors relate to difficulties Chiapas is having during database operations. Generally, they

relate to difficulty to communicating with a database server. This may be related to invalid syntax or improper commands being given to the database server.

ERROR_C005: A null UPDATE string was passed to the parser. There may have been difficulties indexing the source X12 file.
ERROR_C006: The SQL failed (...) failed with this message
ERROR_C007: X12SqlConnection Error
ERROR_C008: X12SqlConnection Open Error
ERROR_C009: X12SqlConnection Close Error
ERROR_C010: X12SqlCommand Error
ERROR_C011: X12SqlCommand ExecuteNonQuery Error
ERROR_C012: X12SqlCommand ExecuteScalar Error
ERROR_C013: X12SqlCommand Dispose
ERROR_C014: X12SqlCommand ExecuteReader
ERROR_C015: X12SqlCommand ExecuteReader
ERROR_C016: X12SqlDataReader Read Error
ERROR_C017: X12SqlDataReader Close
ERROR_C018: X12SqlDataReader GetInt32
ERROR_C019: X12SqlDataReader GetInt64 Error
ERROR_C020: X12SqlDataReader GetString Error
ERROR_C021: X12SqlDataReader IsDBNull Error
ERROR_C022: X12SqlDataReader GetDataTypeName
ERROR_C023: X12SqlDataReader DataToString Error
ERROR_C024: X12SqlDataReader GetName Error

TRANSPORTTEXT MODULE

ERROR_X010: Bad Format Key.
This error occurs during DATAKEY_FROM_FLAT when the format CSV file is invalid or corrupt.

ERROR_X011: No lines detected in input file.
When a DATAKEY_FROM_FLAT is given a blank flat file, this error triggers.

ERROR_X012: FlatTextToDataKey Failure
This is a general catchall error when a DATAKEY_FROM_FLAT fails. The format file may not be correctly defining the incoming flat file.

X12COMMON MODULE

These HIPAA errors relate to problems Chiapas finds when decoding or encoding.

ERROR_H001: Missing element
The Hierarchy Tree marks an element as required and it is missing from the incoming HIPAA file.

ERROR_H002: Invalid element
The Hierarchy Tree marks an element as unused and yet there it is populated with data.

ERROR_H003: Invalid element length
The length for this element is outside the specified minimum and maximum lengths.

ERROR_H004: Invalid element type
The data type for this element is incorrect; for example, an invalid date/time is in a date/time field

type or a non-number is in a numeric element type.

ERROR_H005: 'E' (either but not both) Requirement Condition violated

ERROR_H006: 'R' (at least one needs to be present) Requirement Condition violated

ERROR_H007: 'L' (if first present, 2nd or 3rd is required) Requirement Condition violated

ERROR_H008: 'P' (if one is present, both are needed) Requirement Condition violated

ERROR_H009: 'C' (if first is present, second is required) Requirement Condition violated

ERROR_H010: A required segment was not present in the HIPAA file
A segment listed as required by the HIPAA Implementation Guides for this loop was missing from the incoming file.

ERROR_H011: Mandatory Loop Missing
A loop listed as required by the HIPAA Implementation Guide for this specification was not found by the Chiapas Decoder.

ERROR_H012: The amount of times this loop can be present has been exceeded
The HIGs list a certain number of times a loop can be present. If that amount is exceeded, this error will trigger.

ERROR_H013: A segment was encountered that did not match the X12 specification.
The Chiapas Decoder encountered a segment that does not match the specification. This halts all decoding and results in immediate discard of the file, and is a critical error.

ERROR_H014: Invalid value
A value was supplied for an element that does not belong to a group of values listed for that element.

ERROR_X013: Map Key is a blank with no mappings.
Chiapas encountered an empty Map Key, which is invalid.

ERROR_X014: The provided Map Key references a level greater than 20 or 0. These are not valid levels for Chiapas mappings.
Chiapas limits the amount of levels found in a Map Key to 20. In most cases, two to three levels will suffice.

ERROR_X015: There is a Map Key which contains an invalid name; either an empty string or begins with a non-alphabetic character.
Map Key mapping names must start with a letter.

ERROR_X016: The mapping key name contains invalid characters. Numbers, letters and the underscore are the only ones accepted.
Map Key mapping names must start with a letter and may only contain letters, numbers and the

underscore character.

ERROR_X017: Could not resolve segment reference for mapping key
The path to the specified segment could not be resolved in the loaded Hierarchy Tree.

ERROR_X018: Could not resolve element reference for mapping key
The path to an element in the given segment could not be resolved in the loaded Hierarchy Tree.

ERROR_X019: There are no primary keys found in the mapping key.
Primary Keys are required for both encoding and decoding operations. At least one must be present for every level.

ERROR_X020: There was more than one primary key found for a certain level. Only one element can be a primary key at each level.
A primary key is composed of one field only – multiple fields cannot be used at the same level. Two part primary keys can be represented by splitting the fields into two separate levels. For example, the unique claim ID and line item number make a natural two-part primary key.

ERROR_E001: Mapping and Data source conflict.
Chiapas detected a mismatch between the number of columns between the data source and the mapping key. They must share the exact same number of columns.

ERROR_E002: Critical mapping error. Internal error.
Chiapas found a discrepancy in the number of fields found in both the mapping key and the data source. They must be exactly the same, aside from special “non-data” mapping fields with names beginning with CHIAPAS_.

ERROR_E003: Encoder data invalid, empty dataset.
The incoming Data Key is empty.

Appendix C: SQLite Function Reference

SQLite is documented on its home website at www.sqlite.org . Here is a repeat of the functions available in SQLite syntax, in addition to the Chiapas-specific functions available in version 3.3_rt.

Function	Explanation
`abs(X)`	Return the absolute value of argument X.
`coalesce(X,Y,...)`	Return a copy of the first non-NULL argument. If all arguments are NULL then NULL is returned. There must be at least two arguments.
`glob(X,Y)`	This function is used to implement the "X GLOB Y" syntax of SQLite. The sqlite3_create_function() interface can be used to override this function and thereby change the operation of the GLOB operator.
`ifnull(X,Y)`	Return a copy of the first non-NULL argument. If both arguments are NULL then NULL is returned. This behaves the same as coalesce() above.
`last_insert_rowid()`	Return the ROWID of the last row insert from this connection to the database. This is the same value that would be returned from the sqlite_last_insert_rowid() API function.
`length(X)`	Return the string length of X in characters. If SQLite is configured to support UTF-8, then the number of UTF-8 characters is returned, not the number of bytes.
`like(X,Y [,Z])`	This function is used to implement the "X LIKE Y [ESCAPE Z]" syntax of SQL. If the optional ESCAPE clause is present, then the user-function is invoked with three arguments. Otherwise, it is invoked with two arguments only. The sqlite_create_function() interface can be used to override this function and thereby change the operation of the LIKE operator. When doing this, it may be important to override both the two-argument and three-argument versions of the like() function. Otherwise, different code may be called to implement the LIKE operator, depending on whether or not an ESCAPE clause was specified.
`lower(X)`	Return a copy of string X will all characters converted to lower case. The C library tolower() routine is used for the conversion, which means that this function might not work correctly on UTF-8 characters.
`max(X,Y,...)`	Return the argument with the maximum value. Arguments may be strings in addition to numbers. The maximum value is determined by the usual sort order. Note that max() is a simple function when it has two or more arguments, but converts to an aggregate function if given only a single argument.
`min(X,Y,...)`	Return the argument with the minimum value. Arguments may be strings in addition to numbers. The minimum value is determined by the usual sort order. Note that min() is a simple function when it has two or more arguments but converts to an aggregate function if given only a single argument.
`nullif(X,Y)`	Return the first argument if the arguments are different; otherwise, return NULL.

Function	Explanation
quote(X)	This routine returns a string which is the value of its argument suitable for inclusion into another SQL statement. Strings are surrounded by single quotes, with escapes on interior quotes as needed. BLOBs are encoded as hexadecimal literals. The current implementation of VACUUM uses this function. The function is also useful when writing triggers to implement undo/redo functionality.
random(*)	Return a random integer between -2147483648 and +2147483647.
round(X)	Rounds X to the nearest whole number.
round(X,Y)	Round off the number X to Y digits to the right of the decimal point.
soundex(X)	Compute the soundex encoding of the string X. The string '?000' is returned if the argument is NULL. This function is omitted from SQLite by default. It is only available if the -DSQLITE_SOUNDEX=1 compiler option is used when SQLite is built.
sqlite_version(*)	Return the version string for the SQLite library that is running. Example: '2.8.0'
substr(X,Y,Z)	Return a substring of input string X that begins with the Yth character and which is Z characters long. The leftmost character of X is number 1. If Y is negative, then the first character of the substring is found by counting from the right rather than the left. If SQLite is configured to support UTF-8, then characters indices refer to actual UTF-8 characters, not bytes.
typeof(X)	Return the type of the expression X. The only return values are 'null,' 'integer,' 'real, ' 'text' and 'blob.' SQLite's type handling is explained in Datatypes in SQLite Version 3.
upper(X)	Return a copy of input string X converted to all upper-case letters. The implementation of this function uses the C library routine toupper() which means it may not work correctly on UTF-8 strings.
soft0(X)	Return X with trailing 0s removed, or null if X is 0.
hard0(X)	Return X with trailing 0s removed, or 0 if X is null.
pad0(X,Y)	Returns Y with preceding zeroes padding it to X characters.
newdate(X)	When X is between 16 and 99, returns the digits '19' followed by X. When X is between 0 and 15, returns the digits '20' followed by X. For all other numbers, X is returned unchanged.
replace(X,Y,Z)	Returns Z with any occurrences of X replaced with Y.
to_dotcode(X)	Returns X with a '.' inserted between character positions 3 and 4, unless X begins with an 'E' in which case the dot is inserted between positions 4 and 5, e.g., E8047 = E804.7 or 95901 = 959.01
fx_dat_to_str(X,Y)	Returns the incoming date Y (in format MM/DD/YYYY HH:MM AM/PM) into a different format specified by X. X is composed of: YYYY – 4 digit year YY – 2 digit year MM – month DD – day HH – 24-hour 2 digit hour HA – 12-hour 2 digit hour MI – minute TM – AM/PM qualifier Example: FX_DAT_TO_STR('YYYYMMDD','7/1/2003')='20030701'

Function	Explanation
`fx_str_to_dat(X,Y)`	Returns the Y date, formatted according to the key above via the X argument, in the format MM/DD/YYYY. Example: FX_STR_TO_DAT('YYYYMMDD','20030701')='07/01/2003'
`extract(X,Y,Z)`	Returns an expression inside of X bounded by the Y character at the position defined by Z. Example: EXTRACT('13>A>1','>',2) = 'A'
`trim(X)`	Returns X with preceding and trailing spaces removed.

Aggregate Function	Explanation
`avg(X)`	Return the average value of all X within a group.
`count(X)` `count(*)`	The first form return a count of the number of times that X is not NULL in a group. The second form (with no argument) returns the total number of rows in the group.
`max(X)`	Return the maximum value of all values in the group. The usual sort order is used to determine the maximum.
`min(X)`	Return the minimum non-NULL value of all values in the group. The usual sort order is used to determine the minimum. NULL is only returned if all values in the group are NULL.
`sum(X)`	Return the numeric sum of all values in the group.

Chiapas HIPAA DECODING CHECKLIST

Business Requirements

Who is the recipient?

What is the business transaction? Circle 997 if a response is needed.

834 835 837 I 837 P 820 997

What are their business requirements?

What are your business requirements?

Inventory

_ Create a list of business fields required by your business systems for this transaction
_ Get a sample HIPAA file that represents a typical business transaction
_ Test the sample file via a testing bureau to make sure it contains valid HIPAA syntax and follows valid business rules

Mapping

_ Map all HIPAA fields to internal business fields.
_ Add in new fields for mandatory derived/static field information until all business requirements are met

Data Shaping

_ Decode sample file via Chiapas and establish the baseline raw decode output
_ Create data extract based on the baseline raw decode output; execute data shaping on the fields to transform the data from HIPAA formats into a format acceptable by your business systems

Testing, Release & Production

_ Put data extract into test environment, and make sure the desired business changes are being made
_ If 997 Transaction Acknowledgment files are requested, set up Chiapas to generate them. Test 997 files with submitter.
_ Test all incoming HIPAA files with a testing bureau for 2 months; continue doing syntax checking indefinitely via Chiapas thereafter.

Chiapas HIPAA ENCODING CHECKLIST

Business Requirements

Who is the recipient?

What is the business transaction? Circle 997 if a response is needed.

834 835 837 I 837 P 820 997

What are their business requirements?

What are your business requirements?

Inventory

_ Generate a list of business data that needs to be provided to the recipient
_ Gather the business data sources and fields needed to provide this data to the recipient. Create a field list.

Mapping

_ Map all fields to HIPAA elements.
_ Add in new fields for mandatory derived/static field information until all required and situational element requirements are met.

Data Shaping

_ Revise data extract to shape data fields into HIPAA friendly formats: dates into YYYYMMDD, dollar amounts converted into values without trailing zeroes, etc.
_ Make sure data is sorted according to Business IDs, such as Member ID #'s or Claim ID and Service Line Numbers.
_ Merge all fields together and create a data extract in a form Chiapas can accept: flat files, CSV files, SQL SELECT query, and so on.

Testing, Release & Production

_ Test syntax via Chiapas. Apply fixes as necessary.
_ Execute deep testing via a testing bureau; apply fixes as necessary.
_ Test with business partner until the file is cleared for production.
_ Go live to production.
_ Continue testing with service bureau for two months; continue HIPAA syntax checking on all exports indefinitely.