Master Index Standardization Engine Reference

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 821–0860–10
December 2009

091220@23031

Contents

Master Index Standardization Engine Reference

The topics listed here provide conceptual information about the Master Index Standardization Engine and how it standardizes data in a master index application.

- Master Index Standardization Engine Overview
- Finite State Machine Framework Configuration
- FSM–Based Person Name Configuration
- FSM–Based Telephone Number Configuration
- Rules–Based Address Data Configuration
- Rules-Based Business Name Configuration
- "Custom FSM–Based Data Types and Variants" on page 78

About the Master Index Standardization Engine

The Master Index Standardization Engine works together with the Master Index Match Engine to provide data parsing, data standardization, phonetic encoding, and record matching capabilities for external applications, such as master index applications. Before records can be compared to evaluate the possibility of a match, the data contained in those records must be normalized and in certain cases phonetically encoded or parsed. Once the data is conditioned, the match engine determines a match weight for each field defined for matching. The standardization engine is built on a flexible framework that allows you to customize the standardization process and extend standardization rules.

Related Topics

Several topics provide information and instructions for implementing and using a master index application. For a complete list of topics related to working with the service-enabled version of Sun Master Index, see "Related Topics" in *Sun Master Index User's Guide*.

Master Index Standardization Engine Overview

The Master Index Standardization Engine is designed to work with the master index applications created by Sun Master Index. The standardization engine can also be called from other applications, web services, web applications, and so on. It is highly configurable in the Sun Master Index environment and can be used to standardize various types of data. The Master Index Standardization Engine works in conjunction with the Master Index Match Engine to improve the quality of your data.

The following topics provide information about standardization concepts, the standardization process, and the Master Index Standardization Engine frameworks.

- "Standardization Concepts" on page 8
- "How the Master Index Standardization Engine Works" on page 9
- "Sun Master Index Standardization and Matching Process" on page 14
- "Master Index Standardization Engine Internationalization" on page 15

Standardization Concepts

Data standardization transforms input data into common representations of values to give you a single, consistent view of the data stored in and across organizations. Standardizing the data stored in disparate systems provides a common representation of the data so you can easily and accurately compare data between systems.

Data standardization applies three transformations against the data: parsing into individual components, normalization, and phonetic encoding. These actions help cleanse data to prepare it for matching and searching. Some fields might require all three steps, some just normalization and phonetic conversion, and other data might only need phonetic encoding. Typically data is first parsed, then normalized, and then phonetically encoded, though some cleansing might be needed prior to parsing.

Standardization can include any one or any combination of the following phases.

- "Data Parsing or Reformatting" on page 8
- "Data Normalization" on page 9
- "Phonetic Encoding" on page 9

Data Parsing or Reformatting

If incoming records contain data that is not formatted properly, it must be reformatted before it can be normalized. This process identifies and separates each component of a free-form text field that contains multiple pieces of information. Reformatting can also include removing characters or strings from a field that are not relevant to the data. A good example is standardizing free-form text address fields. If you are comparing or searching on street addresses that are contained in one or more free-form text fields (that is, the street address is contained in one field, apartment number in another, and so on), those fields need to be parsed into their individual components, such as house number, street name, street type, and street direction. Then certain components of the address, such as the street name and type, can be normalized. Field components are also known as tokens, and the process of separating data into its tokens is known as tokenization.

Data Normalization

Normalizing data converts it into a standard or common form. A common use for normalization is to convert nicknames into their standard names, such as converting “Rich” to “Richard” or “Meg” to “Margaret”. Another example is normalizing street address components. For example, both “Dr.” or “Drv” in a street address might be normalized to “Drive”. Normalized values are obtained from lookup tables. Once a field value is normalized, that value can be more accurately compared against values in other records to determine whether they are a match.

Phonetic Encoding

Once data has gone through any necessary reformatting and normalization, it can be phonetically encoded. In a master index application, phonetic values are generally used in blocking queries in order to obtain all possible matches to an incoming record. They are also used to perform searches from the Master Index Data Manager (MIDM) that allow for misspellings and typographic errors. Typically, first names use Soundex encoding and last names and street names use NYSIIS encoding, but the Master Index Standardization Engine supports several additional phonetic encoders as well.

How the Master Index Standardization Engine Works

The Master Index Standardization Engine uses two frameworks to define standardization logic. One framework is based on a finite state machine (FSM) model and the other is based on rules programmed in Java. In the current implementation, the person names and telephone numbers are processed using the FSM framework, and addresses and business names are processed using the rules-based framework. The Master Index Standardization Engine includes several sets of files that define standardization logic for all supported data types. For person data and addresses, one set of standardization files is provided for the following national variants: Australia, France, Great Britain, and the United States. You can customize these files to adapt the standardization and matching logic to your specific needs or you can create new data types or variants for even more customized processing. With pluggable standardization sets, you can define custom standardization processing for most types of data.

The following topics provide information about the Master Index Standardization Engine, the standardization frameworks, and data is standardized:

- “Master Index Standardization Engine Data Types and Variants” on page 10
- “Master Index Standardization Engine Standardization Components” on page 10
- “Finite State Machine Framework” on page 11
- “Rules-Based Framework” on page 12

Master Index Standardization Engine Data Types and Variants

A data type is the primary kind of data you are processing, such as person names, addresses, business names, automotive parts, and so on. A variant is a subset of a data type that is designed to standardize a specific kind of data. For example, for addresses and names, the variants typically define rules for the different countries in which the data originates. For automotive parts, the variants might be different manufacturers. Each data type and variant uses its own configuration files to define how fields in incoming records are parsed, standardized, and classified for processing. Data types are sometimes referred to as *standardization types*.

In the default implementation with a master index application, the engine supports data standardization on the following types of data:

- Person Information (described in FSM–Based Person Name Configuration)
- Telephone Numbers (described in FSM–Based Telephone Number Configuration)
- Street Addresses (described in Rules–Based Address Data Configuration)
- Business Names (described in Rules-Based Business Name Configuration)

In the default configuration, the standardization engine expects street address and business names to be in free-form text fields that need to be parsed prior to normalization and phonetic encoding. Person and phone information can also be contained in free-form text fields, but theses types of information can also be processed if the data is already parsed into its individual components. Each data type requires specific customization to mefa.xml in the master index project. This can be done by modifying the file directly or by using the Master Index Configuration Editor.

Master Index Standardization Engine Standardization Components

The Master Index Standardization Engine breaks down fields into various components during the parsing process. This is known an tokenization. For example, it breaks addresses into floor number, street number, street name, street direction, and so on. Some of these components are similar and might be stored in the same output field. In the default configuration for a master index application, for example, when the standardization engine finds a house number, rural route number, or PO box number, the value is stored in the HouseNumber database field. You can customize this as needed, as long as any field you specify to store a component is also included in the object structure defined for the master index application.

The standardization engine uses tokens to determine how to process fields that are defined for normalization or parsing into their individual standardization components. For FSM-based data types, the tokens are defined as output symbols in the process definition files and are referenced in the standardization structures in the Master Index Configuration Editor and in mefa.xml. The tokens determine how each field is normalized or how a free-form text field is parsed and normalized. For rules-based data types, the tokens are defined internally in the Java code. The tokens for business names specify which business type key file to use to normalize a specific standardization component. The tokens for addresses determine which database fields store each standardization component and how each component is standardized.

Finite State Machine Framework

A finite state machine (FSM) is composed of one or more states and the transitions between those states. The Master Index Standardization Engine FSM framework is designed to be highly configurable and can be easily extended with no Java coding. The following topics describe the FSM framework and the configuration files that define FSM–based standardization.

About the Finite State Machine Framework

In an FSM framework, the standardization process is defined as one or more states. In a state, only the input symbols defined for that state are recognized. When one of those symbols is recognized, the following action or transition is based on configurable processing rules. For example, when an input symbol is recognized, it might be preprocessed by removing punctuation, matched against a list of tokens, and then postprocessed by normalizing the input value. Once this has been completed for all input symbols, the standardization engine determines which token is the most likely match.

FSM-based processing includes the following steps:

- **Cleansing** – The entire input string is modified to make sure it is broken down into its individual components correctly.
- **Tokenization** – The input string is broken down into its individual components.
- **Parsing** – The individual field components are processed according to configurable rules. Parsing can include any combination of the following three stages:
  - **Preprocessing** – Each token is cleansed prior to matching to make the value more uniform.
  - **Matching** – The cleansed token is matched against patterns or value lists.
  - **Postprocessing** – The matched token is normalized.

Note – Several parsing sequences might be performed against one field component in order to best match it with a token. Each sequence is carried out until a match is made.

- **Ambiguity Resolution** – Some input strings might match more than one processing rule, so the FSM framework includes a probability-based mechanism for determining the correct state transition.

Using the person data type, for example, first names such as “Bill” and “Will” are normalized to “William”, which is then converted to its phonetic equivalent. Standardization logic is defined in the standardization engine configuration files and in the Master Index Configuration Editor or mefa.xml in a master index project.

FSM-Based Configuration

The FSM-based standardization configuration files are stored in the master index project and appear in the Standardization Engine node of the project. These files are separated into groups based on the primary data types being processed. Data type groups have further subsets of configuration files based on the variants for each data type. FSM-based data types and variants, such as PersonName and PhoneNumber, include the following configuration file types.

- **Service Definition Files** – Each data type and data type variant is defined by a service definition file. Service *type* files define the fields to be standardized for a data type and service *instance* files define the variant and Java factory class for the variant. Both files are in XML format and should not be modified unless the data type is extended to include more output symbols.
- **Process Definition Files** – These files define the different stages of processing data for the data type or variant. It defines the FSM states, input and output symbols, patterns, and data cleansing rules. These files use a domain-specific language (DSL) to define how the data fields are processed.
- **Lexicon Files** – The standardization engine uses these files to recognize input data. A lexicon provides a list of possible values for a specific field, and one lexicon file should be defined for each field on which standardization is performed.
- **Normalization Files** – The standardization engine uses these files to convert nonstandard values into a common form. For example, a nickname file provides a list of nicknames along with the common version of each name. For example, “Beth” and “Liz” might both be normalized to “Elizabeth”. Each row in the file contains a nickname and its corresponding normalized version separated by a pipe character (|).

Rules-Based Framework

In the rules-based framework, the standardization process is define in the underlying Java code. You can configure several aspects of the standardization process, such as the detectable patterns for each data type, how values are normalized, and how the input string is cleansed and parsed. You can define custom rules-based data types and variants by creating custom Java packages that define processing.

About the Rules-Based Framework

In the rules-based framework, individual field components are recognized by the patterns defined for each data type and by information provided in configurable files about how to preprocess, match, and postprocess each field components. The rules-based framework processes data in the following stages.

- **Parsing** - A free-form text field is separated into its individual components, such as street address information or a business name. This process takes into account logic you can customize, such as token patterns, special characters, and priority weights for patterns.
- **Normalization** - Once a field is parsed, individual components of the field are normalized based on the configuration files. This can include changing the input street name to a common form or changing the input business name to its official form.
- **Phonetic Encoding** - After a field is parsed and optionally normalized, the value of a field is converted to its phonetic version. The value to be converted can be the original input value, a parsed value, a normalized value, or a parsed and normalized value.

Using the street address data type, for example, street addresses are parsed into their component parts, such as house numbers, street names, and so on. Certain fields are normalized, such as street name, street type, and street directions. The street name is then phonetically converted. Standardization logic is defined in the standardization engine configuration files and in the Master Index Configuration Editor or mefa.xml in a master index project.

Rules-Based Configuration

The rules-based standardization configuration files are stored in the master index project and appear as nodes in the Standardization Engine node of the project. These files are separated into groups based on the primary data types and variants being processed. Rules-based data types and variants, such as the default Address and Business Name types, use the following configuration file types.

- **Service Definition Files** – Each data type and data type variant is configured by a service definition file. Service type files define the fields to be standardized for a data type, and service instance definition files define the variant and Java factory class for the variant. Both files are in XML format. These files should not be modified.
- **Category Files** - The standardization engine uses category files when processing business names. These files list common values for certain types of data, such as industries and organizations for business names. Category files also define standardized versions of each term or classify the terms into different categories, and some files perform both functions. When processing address files, category files named *clues files* are used.

- **Clues Files** - The standardization engine uses clues files when processing address data types. These files list general terms used in street address fields, define standardized versions of each term, and classify the terms into various component types using predefined address tokens. These files are used by the standardization engine to determine how to parse a street address into its various components. Clues files provide clues in the form of tokens to help the engine recognize the component type of certain values in the input fields.
- **Patterns Files** - The patterns files specify how incoming data should be interpreted for standardization based on the format, or pattern, of the data. These files are used only for processing data contained in free-form text fields that must be parsed prior to matching (such as street address fields or business names). Patterns files list possible input data patterns, which are encoded in the form of tokens. Each token signifies a specific component of the free-form text field. For example, in a street address field, the house number is identified by one token, the street name by another, and so on. Patterns files also define the format of the output fields for each input pattern.
- **Key Type Files** - For business name processing, the standardization engine refers to a number of key type files for processing data. These files generally define standard versions of terms commonly found in business names and some classify these terms into various components or industries. These files are used by the standardization engine to determine how to parse a business name into its different components and to recognize the component type of certain values in the input fields.
- **Reference Files** - Reference files define general terms that appear in input fields for each data type. Some reference files define terms to ignore and some define terms that indicate the business name is continuing. For example, in business name processing “and” is defined as a joining term. This helps the standardization engine to recognize that the primary business name in “Martin and Sons, Inc.” is “Martin and Sons” instead of just “Martin”. Reference files can also define characters to be ignored by the standardization engine.

Sun Master Index Standardization and Matching Process

In a default Sun Master Index implementation, the master index application uses the Master Index Match Engine and the Master Index Standardization Engine to cleanse data in real time. The standardization engine uses configurable pattern-matching logic to identify data and reformat it into a standardized form. The match engine uses a matching algorithm with a proven methodology to process and weight records in the master index database. By incorporating both standardization and matching capabilities, you can condition data prior to matching. You can also use these capabilities to review legacy data prior to loading it into the database. This review helps you determine data anomalies, invalid or default values, and missing fields.

In a master index application, both matching and standardization occur when two records are analyzed for the probability of a match. Before matching, certain fields are normalized, parsed, or converted into their phonetic values if necessary. The match fields are then analyzed and weighted according to the rules defined in a match configuration file. The weights for each field are combined to determine the overall matching weight for the two records. After these steps are complete, survivorship is determined by the master index application based on how the overall matching weight compares to the duplicate and match thresholds of the master index application.

In a master index application, the standardization and matching process includes the following steps:

1. The master index application receives an incoming record.
2. The Master Index Standardization Engine standardizes the fields specified for parsing, normalization, and phonetic encoding. These fields are defined in mefa.xml and the rules for standardization are defined in the standardization engine configuration files.
3. The master index application queries the database for a candidate selection pool (records that are possible matches) using the blocking query specified in master.xml. If the blocking query uses standardized or phonetic fields, the criteria values are obtained from the database.
4. For each possible match, the master index application creates a match string (based on the match columns in mefa.xml) and sends the string to the Master Index Match Engine.
5. The Master Index Match Engine checks the incoming record against each possible match, producing a matching weight for each. Matching is performed using the weighting rules defined in the match configuration file.

Master Index Standardization Engine Internationalization

By default, the Master Index Standardization Engine is configured for addresses and names originating from Australia, France, Great Britain, and the United States, and for telephone numbers and business names of any origin. Each national variant for each data type uses a specific subset of configuration files. In addition, you can define custom national variants for the standardization engine to support addresses and names from other countries and to support other data types. You can process with your data using the standardization files for a single variant or you can use multiple variants depending on how the master index application is configured.

Finite State Machine Framework Configuration

In the FSM framework, the state model definition, along with all the token processing logic, is provided in configuration files in XML format. In addition, lexicon and normalization files define logic used by the Master Index Standardization Engine to recognize and normalize specific values for each data type or variant. The standardization configuration files for the Master Index Standardization Engine must follow certain rules for formatting and interdependencies. The following topics provide an overview of the types of configuration files provided for standardization.

- “FSM Framework Configuration Overview” on page 16
- “Process Definition File” on page 17
- “Lexicon Files” on page 29
- “Normalization Files” on page 30

FSM Framework Configuration Overview

The configuration of the finite state machine (FSM) includes defining the various states, transitions between those states, and any actions to perform during each state. Each instance of the FSM begins in the *start* state. In each state, the standardization engine looks for the next token (or *input symbol*), optionally performs certain actions against the token, determines the potential output symbols, and then uses probability-based logic to determine the output symbol to generate for the state and how to transition to the next state. Within each state, only the input symbols defined for that state are recognized. When an input symbol is recognized, the processing defined for that symbol is carried out and the transition to the next state occurs. Note that some input symbols might trigger a transition back to the current state. Once the standardization engine does not recognize any input symbols, the FSM reaches a terminal state from which no further transitions are made.

You can define specialized processing rules for each input symbol in the state model. These rules include cleansing and data transformation logic, such as converting data to uppercase, removing punctuation, comparing the input value against a list of values, and so on. Both the state model and the processing rules are defined in the process definition file, `standardizer.xml`. The lists that you can use to compare and normalize values for each input symbol are contained in lexicon and normalization files.

The configuration files that configure the standardization engine are stored in the master index project and appear as nodes in the Standardization Engine node of the project. The standardization files are separated into subsets that are each unique to a specific data type, which are further grouped into variants on those data types. You can define additional standardization file subsets to create new variants or even create new data types, such as automotive parts, inventory items, and so on.

The following topics provide information about the files you can configure or create to customize how your data is standardized:

- “Process Definition File” on page 17
- “Lexicon Files” on page 29
- “Normalization Files” on page 30

Process Definition File

The process definition file (`standardizer.xml`) is the primary configuration file for standardization. It defines the state model, input and output symbol definitions, preprocessing and postprocessing rules, and normalization rules for any type of standardization. Using a domain-specific markup language, you can configure any type of standardization without having to code a new Java package. Each process definition file defines the different stages of processing data for one data type or variant. The process definition file is stored in the `resource` folder under the data type or variant it defines.

The process definition file is divided into six primary sections, which are described in the following topics:

- "Standardization State Definitions" on page 17
- "Input Symbol Definitions" on page 19
- "Output Symbol Definitions" on page 21
- "Data Cleansing Definitions" on page 22
- "Data Normalization Definitions" on page 23
- "Standardization Processing Rules Reference" on page 24

The processing flow is defined in the state definitions. The input symbol definitions specify the token preprocessing, matching, and postprocessing logic. This is the logic carried out for each input token in a given state. The output symbols define the output for each state. The data cleansing definitions specify any transformations made to the input string prior to tokenization. Normalization definitions are used for data that does not need to be tokenized, but only needs to be normalized and optionally phonetically encoded. For example, if the input text provides the first name in its own field, the middle name in its own field, and so on, then only the normalization definitions are used to standardize the data. The standardization processing rules can be used in all sections except the standardization state definitions.

Standardization State Definitions

An FSM framework is defined by its different states and transitions between states. Each FSM begins with a start state when it receives an input string. The first recognized input symbol in the input string determines the next state based on customizable rules defined in the state model section of `standardizer.xml`. The next recognized input symbol determines the transition to the next state. This continues until no symbols are recognized and the termination state is reached.

Below is an excerpt from the state definitions for the PersonName data type. In this state, the first name has been processed and the standardization engine is looking for one of the following: a first name (indicating a middle name), a last name, an abbreviation (indicating a middle initial), a conjunction, or a nickname. A probability is given for each of these symbols indicating how likely it is to be the next token.

```
<stateModel name="start">
   <when inputSymbol="salutation" nextState="salutation"
         outputSymbol="salutation" probability=".15"/>
   <when inputSymbol="givenName" nextState="headingFirstName"
         outputSymbol="firstName" probability=".6"/>
   <when inputSymbol="abbreviation" nextState="headingFirstName"
         outputSymbol="firstName" probability=".15"/>
   <when inputSymbol="surname" nextState="trailingLastName"
         outputSymbol="lastName" probability=".1"/>
   <state name="headingFirstName">
      <when inputSymbol="givenName" nextState="headingMiddleName"
            outputSymbol="middleName" probability=".4"/>
      <when inputSymbol="surname" nextState="headingLastName"
            outputSymbol="lastName" probability=".3"/>
      <when inputSymbol="abbreviation" nextState="headingMiddleName"
            outputSymbol="middleName" probability=".1"/>
      <when inputSymbol="conjunction" nextState="headingFirstName"
            outputSymbol="conjunction" probability=".1"/>
      <when inputSymbol="nickname" nextState="firstNickname"
            outputSymbol="nickname" probability=".1"/>
   </state>
   ...
```

The following table lists and describes the XML elements and attributes for the standardization state definitions.

Element	Attribute	Description
stateModel		The primary container element for the state model that includes the definitions for each state in the FSM. This element contains a series of *when* elements as described below to define the transitions from the start element to any of the other states. It also contains a series of *state* elements that define the remaining FSM states.
	name	The name of start state (by default, “start”).
state		A definition for one state in the FSM (not including the start state). Each state element contains a series of *when* elements and attributes as described above to define the processing flow.
	name	The name of the state. The names defined here are referenced in the *nextState* attributes described below to specify the next state.
when		A statement defining which state to transition to and which symbol to output when a specific input symbol is recognized in each state. These elements define the possible transitions from one state to another.

Element	Attribute	Description
	inputSymbol	The name of an input symbol that might occur next in the input string. This must match one of the input symbols defined later in the file. For more information about input symbols and their processing logic, see "Input Symbol Definitions" on page 19.
	nextState	The name of the next state to transition to when the specified input symbol is recognized. This must match the name of one of the states defined in the state model section.
	outputSymbol	The name of the symbol that the current state produces for when processing is complete for the state based on the input symbol. Not all transitions have an output symbol. This must match one of the output symbols defined later in the file. For more information, see "Output Symbol Definitions" on page 21
	probability	The probability that the given input symbol is actually the next symbol in the input string. Probabilities are indicated by a decimal between and including 1 and 0. All probabilities for a given state must add up to 1. If a state definition includes the *eof* element described below, all probabilities including the *eof* probability must add up to 1.
eof	probability	The probability that the FSM has reached the end of the input string in the current state. Probabilities are indicated by a decimal between and including 1 and 0. The sum of this probability and all other probabilities for a given state must be 1.

Input Symbol Definitions

The input symbol definitions name and define processing logic for each input symbol recognized by the states. For each state, each possible input symbol is tried according to the rules defines here, and then the probability that it is the next token is assessed. Each input symbol might be subject to preprocessing, token matching, and postprocessing. Preprocessing can include removing punctuation or other regular expression substitutions. The value can then be matched against values in the lexicon file or against regular expressions. If the value matches, it can then be normalized based on the specified normalization file or on pattern replacement. One input symbol can have multiple preprocessing, matching, and postprocessing iterations to go through. If their are multiple iterations, each is carried out in turn until a match is found. All of these steps are optional.

Below is an excerpt from the input symbol definitions for PersonName processing. This excerpt processes the salutation portion of the input string by first removing periods, then comparing the value against the entries in the `salutation.txt` file, and finally normalizing the matched value based on the corresponding entry in the `salutationNormalization.txt` file. For example, if the value to process is "Mr.", it is first changed to "Mr", matched against a list of salutations, and then converted to "Mister" based on the entry in the normalization file.

```
<inputSymbol name="salutation">
   <matchers>
      <matcher>
         <preProcessing>
            <replaceAll regex="\." replacement=""/>
         </preProcessing>
         <lexicon resource="salutation.txt"/>
         <postProcessing>
            <dictionary resource="salutationNormalization.txt" separator="\|"/>
         </postProcessing>
      </matcher>
   </matchers>
</inputSymbol>
```

The following table lists and describes the XML elements and attributes for the input symbol definitions.

Element	Attribute	Description
inputSymbol		A container element for the processing logic for one input symbol.
	name	The name of the input symbol against which the following logic applies.
matchers		A list of processing definitions, each of which define one preprocessing, matching, and postprocess sequence. Not all definitions include all three steps.
matcher		A processing definition for one sequence of preprocessing, matching, and postprocessing. A processing definition might contain only one or any combination of the three steps.
	factor	A factor to apply to the probability specified for the input symbol in the state definition. For example, if the state definition probability is .4 and this factor is .25, then the probability for this matching sequence is .1. Only define this attribute when the probability for this matching sequence is very low.
preProcessing		A container element for the preprocessing rules to be carried out against an input symbol. For more information about the rules you can use, see "Standardization Processing Rules Reference" on page 24.

Element	Attribute	Description
lexicon	resource	The name of the lexicon file containing the list of values to match the input symbol against. **Note –** You can also match against patterns or regular expressions. For more information, see `matchAllPatterns` and `pattern` in "Standardization Processing Rules Reference" on page 24.
postProcessing		A container element for the postprocessing rules to be carried out against an input symbol that has been matched. For more information about the rules you can use, see "Standardization Processing Rules Reference" on page 24.

Output Symbol Definitions

The output symbol definitions name each output symbol that can be produced by the defined states. This section can define additional processing for output symbols using the rules described in "Standardization Processing Rules Reference" on page 24. Each output symbol defined in the state model definitions must match a value defined here. Below is an excerpt from the output symbol definitions for PersonName processing.

```
<outputSymbols>
   <outputSymbol name="salutation"/>
   <outputSymbol name="firstName"/>
   <outputSymbol name="middleName"/>
   <outputSymbol name="nickname"/>
   <outputSymbol name="lastName"/>
   <outputSymbol name="generation"/>
   <outputSymbol name="title"/>
   <outputSymbol name="conjunction"/>
</outputSymbols>
```

The following table lists and describes the XML elements and attributes for the output symbol definitions.

Element	Attribute	Description
outputSymbols		A list of output symbols for each processing state.
outputSymbol		A definition for one output symbol.
	name	The name of the output symbol

Element	Attribute	Description
occurrenceConcatenator		An optional class to specify the character that separates contiguous occurrences of the same output symbol. For example, this is used in the PhoneNumber data type to concatenate phone number components that are separated by dashes. Components are concatenated using blanks.
	class	The name of the occurrence concatenator class. One concatenator class is predefined.
property		A parameter for the occurrence concatenator class. For the default class, the parameter specifies a separator character.
	name	The name of the parameter. For the default class, the name is “separator”.
	value	The parameter value.
tokenConcatenator		An optional class to specify the character that separates non-contiguous occurrences of the same output symbol. For example, this is used in the PhoneNumber data type to concatenate phone number components.
	class	The name of the token concatenator class. one concatenator class is predefined.
property		A parameter for the token concatenator class. For the default class, the parameter specifies a separator character.
	name	The name of the parameter. For the default class, the name is “separator”.
	value	The value of the parameter.

Data Cleansing Definitions

You can define cleansing rules to transform the input data prior to tokenization to make the input record uniform and ensure the data is correctly separated into its individual components. This standardization step is optional.

Common data transformations include the following:

- Converting a string to all uppercase.
- Trimming leading and trailing white space.
- Converting multiple spaces in the middle of a string to one space.
- Transliterating accent characters or diacritical marks.
- Adding a space on either side of extra characters (to help the tokenizer recognize them).
- Removing extraneous content.

- Fixing common typographical errors.

The cleansing rules are defined within a *cleanser* element in the process definition file. You can use any of the rules defined in "Standardization Processing Rules Reference" on page 24 to cleanse the data. Cleansing attributes use regular expressions to define values to find and replace.

The following excerpt from the PhoneNumber data type does the following to the input string prior to processing:

- Converts all characters to upper case.
- Replaces the specified input patterns with new patterns.
- Removes white space at the beginning and end of the string and concatenates multiple consecutive spaces into one space.

```
<cleanser>
   <uppercase/>
   <replaceAll regex="([0-9]{3})([0-9]{3})([0-9]{4})" replacement="($1)$2-$3"/>
   <replaceAll regex="([-(),])" replacement=" $1 "/>
   <replaceAll regex="\+(\d+) -" replacement="+$1-"/>
   <replaceAll regex="E?X[A-Z]*[.#]?\s*([0-9]+)" replacement="X $1"/>
   <normalizeSpace/>
</cleanser>
```

Data Normalization Definitions

If the data you are standardizing does not need to be parsed, but does require normalization, you can define data normalization rules to be used instead of the state model defined earlier in the process definition file. These rules would be used in the case of person names where the field components are already contained in separate fields and do no need to be parsed. In this case, the standardization engine processes one field at a time according to the rules defined in the normalizer section of `standardizer.xml`. In this section, you can define preprocessing rules to be applied to the fields prior to normalization.

Below is an excerpt from the PersonName data type. These rules convert the input string to all uppercase, and then processes the FirstName and MiddleName fields based on the givenName input symbol and processes the LastName field based on the surname input symbol.

```
 <normalizer>
   <preProcessing>
      <uppercase/>
   </preProcessing>
   <for field="FirstName" use="givenName"/>
   <for field="MiddleName" use="givenName"/>
   <for field="LastName" use="surname"/>
</normalizer>
```

The following table lists and describes the XML elements and attributes for the normalization definitions.

Element	Attribute	Description
normalizer		A container element for the normalization rules to use when field components do not require parsing, but do require normalization.
preProcessing		A container element for any preprocessing rules to apply to the input strings prior to normalization. For more information about preprocessing rules, see "Standardization Processing Rules Reference" on page 24.
for		The input symbol to use for a given field. This is defined in the following attributes.
	field	The name of a field to be normalized.
	use	The name of the input symbol to associate with the field. The processing logic defined for the input symbol earlier in the file is used to normalize the data contained in that field.

Standardization Processing Rules Reference

The Master Index Standardization Engine provides several matching and transformation rules for input values and patterns. You can add or modify any of these rules in the existing process definition files (`standardizer.xml`). Several of these rules use regular expressions to define patterns and values. See the Javadoc for `java.util.regex` for more information about regular expressions.

The available rules include the following:

- "dictionary" on page 24
- "fixedString" on page 25
- "lexicon" on page 26
- "normalizeSpace" on page 26
- "pattern" on page 27
- "replace" on page 28
- "replaceAll" on page 28
- "transliterate" on page 28
- "uppercase" on page 29

dictionary

This rule checks the input value against a list of values in the specified normalization file, and, if the value is found, converts the input value to its normalized value. This generally used for postprocessing but can also be used for preprocessing tokens. The normalization files are located in the same directory as the process definition file (the `instance` folder for the data type or variant).

The syntax for `dictionary` is:

```
<dictionary resource="file_name" separator="delimiter"/>
```

The parameters for `dictionary` are:

- *resource* – The name of the normalization file to use to look up the input value and determine the normalized value.
- *separator* – The character used in the normalization file to separate the input value entries from the normalized versions. The default normalization files all use a pipe (|) as a separator.

EXAMPLE 1 Sample dictionary Rule

The following sample checks the input value against the list in the first column of the `givenNameNormalization.txt` file, which uses a pipe symbol (|) to separate the input value from its normalized version. When a value is matched, the input value is converted to its normalization version.

```
<dictionary resource="givenNameNormalization.txt" separator="\|"/>
```

fixedString

This rule checks the input value against a fixed value. This is generally used for the token matching step for input symbol processing. You can define a list of fixed strings for an input symbol by enclosing multiple *fixedString* elements within a *fixedStrings* element. The syntax for `fixedString` is:

```
<fixedString>string</fixedString>
```

The parameter for `fixedString` is:

- *string* – The fixed value to compare the input value against.

EXAMPLE 2 Sample fixedString Rules

The following sample matches the input value against the fixed values “AND”, “OR” and “AND/OR”. If one of the fixed values matches the input string, processing is continued for that matcher definition. If no fixed values match the input string, processing is stopped for that matcher definition and the next matcher definition is processed (if one exists).

```
<fixedStrings>
   <fixedString>AND</fixedString>
   <fixedString>OR</fixedString>
```

EXAMPLE 2 Sample fixedString Rules *(Continued)*

```
   <fixedString>AND/OR</fixedString>
</fixedStrings>
```

lexicon

This rule checks the input value against a list of values in the specified lexicon file. This generally used for token matching. The lexicon files are located in the same directory as the process definition file (the `instance` folder for the data type or variant).

The syntax for `lexicon` is:

`<lexicon resource="`*file_name*`/>`

The parameter for `lexicon` is:

- *resource* – The name of the lexicon file to use to look up the input value to ensure correct tokenization.

EXAMPLE 3 Sample lexicon Rule

The following sample checks the input value against the list in the `givenName.txt` file. When a value is matched, the standardization engine continues to the postprocessing phase if one is defined.

```
<lexicon resource="givenName.txt"/>
```

normalizeSpace

This rule removes leading and trailing white space from a string and changes multiple spaces in the middle of a string to a single space. The syntax for `normalizeSpace` is:

```
<normalizeSpace/>
```

EXAMPLE 4 Sample normalizeSpace Rule

The following sample removes the leading and trailing white space from a last name field prior to checking the input value against the `surnames.txt` file.

```
<matcher>
   <preProcessing>
     <normalizeSpace/>
   </preProcessing>
   <lexicon resource="surnames.txt"/>
</matcher>
```

pattern

This rule checks the input value against a specific regular expression to see if the patterns match. You can define a sequence of patterns by including them all in order in a *matchAllPatterns* element. You can also specify sub-patterns to exclude. The syntax for `pattern` is:

```
<pattern regex="regex_pattern"/>
```

The parameter for `pattern` is:

- *regex* – A regular expression to validate the input value against. See the Javadocs for `java.util.regex` for more information.

The `pattern` rule can be further customized by adding `exceptFor` rules that define patterns to exclude in the matching process. The syntax for `exceptFor` is:

```
<pattern regex="regex_pattern"/>
   <exceptFor regex="regex_pattern"/>
</pattern>
```

The parameter for `exceptFor` is:

- *regex* – A regular expression to exclude from the pattern match. See the Javadocs for `java.util.regex` for more information.

EXAMPLE 5 Sample pattern Rule

The following sample checks the input value against the sequence of patterns to see if the input value might be an area code. These rules specify a pattern that matches three digits contained in parentheses, such as (310).

```
<matchAllPatterns>
   <pattern regex="regex="\("/>
   <pattern regex="regex="\[0-9]{3}"/>
   <pattern regex="regex="\)"/>
</matchAllPatterns>
```

The following sample checks the input value to see if its pattern is a series of three letters excluding THE and AND.

```
<pattern regex="[A-Z]{3}">
   <exceptFor regex="regex="THE"/>
   <exceptFor regex="regex="AND"/>
</matchAllPatterns>
```

replace

This rule checks the input value for a specific pattern. If the pattern is found, it is replaced by a new pattern. This rule only replaces the first instance it finds of the pattern. The syntax for `replace` is:

```
<replace regex="regex_pattern" replacement="regex_pattern"/>
```

The parameters for `replace` are:

- *regex* – A regular expression that, if found in the input string, is converted to the replacement expression.
- *replacement* – The regular expression that replaces the expression specified by the *regex* parameter.

EXAMPLE 6 Sample replace Rule

The following sample tries to match the input value against “ST.”. If a match is found, the standardization engine replaces the value with “SAINT”.

```
<replace regex="ST\." replacement="SAINT"/>
```

replaceAll

This rule checks the input value for a specific pattern. If the pattern is found, all instances are replaced by a new pattern. The syntax for `replaceAll` is:

```
<replaceAll regex="regex_pattern" replacement="regex_pattern"/>
```

The parameters for `replaceAll` are:

- *regex* – A regular expression that, if found in the input string, is converted to the replacement expression.
- *replacement* – The regular expression that replaces the expression specified by the *regex* parameter.

EXAMPLE 7 Sample replaceAll Rule

The following sample finds all periods in the input value and converts them to blanks.

```
<replaceAll regex="\." replacement=""/>
```

transliterate

This rule converts the specified characters in the input string to a new set of characters, typically converting from one alphabet to another by adding or removing diacritical marks. The syntax for `transliterate` is:

```
<transliterate from="existing_char" to="new_char"/>
```

The parameters for `transliterate` are:

- *from* – The characters that exist in the input string that need to be transliterated.
- *to* – The characters that will replace the above characters.

EXAMPLE 8 Sample transliterate Rule

The following sample converts lower case vowels with acute accents to vowels with no accents.

```
<transliterate from="áéíóú" to="aeiou"/>
```

uppercase

This rule converts all characters in the input string to upper case. The rule does not take any parameters. The syntax for `uppercase` is:

```
<uppercase/>
```

EXAMPLE 9 Sample uppercase Rule

The following sample converts the entire input string into uppercase prior to doing any pattern or value replacements. Since this is defined in the cleanser section, this is performed prior to tokenization.

```
<cleanser>
   <uppercase/>
   <replaceAll regex="\." replacement=". "/>
   <replaceAll regex="AND / OR" replacement="AND/OR"/>
   ...
</cleanser>
```

Lexicon Files

Lexicon files list the possible values for a specific field that the standardization engine uses to recognize input data. A lexicon file can be defined for each field on which standardization is performed. These files are referenced from the process definition file when defining matching or processing rules. The lexicon files are located in the `resource` folder for the data type or variant from which they are referenced.

Lexicon files are simply text files with a single column that lists the possible field values. They are typically given the same name as the token type, or standardization component, that they define. For example, the lexicon files for first and last names are `givenNames.txt` and `surnames.txt`. You can modify these files as needed to suit your data requirements and you can create new lexicon files to reference from the process definition file.

Below is an excerpt of the given names lexicon file:

```
ALIA
ALICA
ALICAI
ALICE
ALICEMARIE
ALICEN
ALICIA
ALICJA
ALID
ALIDA
ALIHAN
ALINA
ALINE
ALIS
ALISA
ALISE
ALISHA
ALISHIA
ALISIA
ALISON
```

Normalization Files

Normalization files list nonstandard values for a field along with their corresponding normalized value. The standardization engine uses these files to convert nonstandard values into a standard form. These files are referenced from the process definition file when defining normalization rules. The normalization files are located in the `resource` folder for the data type or variant from which they are referenced.

The most common example of normalization is a nickname file that provides a list of nicknames along with the standard version of each name. For example, “Beth” and “Liz” might both be standardized to “Elizabeth”. Each row in the file contains a nickname and its corresponding standardized version separated by a pipe character (|). You can modify these files as needed to suit your data processing needs, or you can create new normalization files to reference from the process definition file.

Below is an excerpt of the given names normalization file:

```
BEV|BEVERLY
BIANCA|BLANCHE
BILLIE|WILLIAM
BILLYE|WILLIAM
BILLY|WILLIAM
BILL|WILLIAM
```

```
BIRGIT|BRIDGET
BLANCA|BLANCHE
BLANCH|BLANCHE
BOBBIE|ROBERT
BOBBI|ROBERT
BOBBYE|ROBERT
BOBBY|ROBERT
BOB|ROBERT
BONNY|BONNIE
BRADLY|BRADLEY
```

FSM–Based Person Name Configuration

By default, person name data is standardized using the finite state machine (FSM) framework. Processing person data might involve parsing free-form data fields, but normally involves normalizing and phonetically encoding certain fields prior to matching. The following topics describe the default configuration that defines person processing logic and provide information about modifying mefa.xml in a master index application for processing person data.

- "Person Name Standardization Overview" on page 31
- "Person Name Standardization Components" on page 32
- "Person Name Standardization Files" on page 32
- "Person Name Standardization and Sun Master Index" on page 34

Person Name Standardization Overview

Processing data with the PersonName data type includes standardizing and matching a person's demographic information. The Master Index Standardization Engine can the create the parsed, normalized, and phonetic values for person data. These values are needed for accurate searching and matching on person data. Several configuration files designed specifically to handle person data are included to provide processing logic for the standardization and phonetic encoding process. The Master Index Standardization Engine can phonetically encode any field you specify.

In addition, when processing person information, you might want to standardize addresses to enable searching against address information. This requires working with the address configuration files described in Rules–Based Address Data Configuration.

Person Name Standardization Components

Standardization engines use tokens to determine how each field is standardized into its individual field components and to determine how to normalize a field value. Tokens also identify the field components to external applications like a master index application. The following table lists each token generated by the Master Index Standardization Engine for person data along with the standardization component they represent. These correspond to the output symbols in the process definition file and to the output fields listed in the service type definition file. For names, you can only specify the predefined field IDs that are listed in this table unless you customize an existing variant or create a new one.

TABLE 1 Person Name Tokens

Token	Description
firstName	Represents a first name field.
generation	Represents a field containing generational information, such as Junior, II, or 3rd.
lastName	Represents a last name field.
middleName	Represents a middle name field.
nickname	Represents a nickname field.
salutation	Represents a field containing prefix information for a name, such as Mr., Miss, or Mrs.
title	Represents a field containing a title, such as Doctor, Reverend, or Professor.

Person Name Standardization Files

Several configuration files are used to define standardization logic for processing person names. You can customize any of the configuration files described in this section to fit your processing and standardization requirements for person data. There are three types of standardization files for person data: process definition, lexicon, and normalization. Four default variants on the PersonName data type are provided that are specialized for standardizing data from France, Australia, the United Kingdom, or the United State. In a master index project, these files appear under PersonName in the Standardization Engine node. Files for each variant appear within sub-folders of PersonName and each corresponds to a specific national variant.

You can customize these files to add entries of other nationalities or languages, including those containing diacritical marks. You can also create new variants to process data of other nationalities. For more information, see Custom Data Types and Variants.

The following topics provide information about each type of person name standardization file:

- "Person Name Lexicon Files" on page 33
- "Person Name Normalization Files" on page 33
- "Person Name Process Definition Files" on page 34

Person Name Lexicon Files

Each PersonName variant contains a set of lexicon files. Each lexicon file contains a list of possible values for a field. The standardization engine matches input values against the values listed in these files to recognize input symbols and ensure correct tokenization. The Master Index Standardization Engine uses these files when processing input symbols as defined in the process definition file (`standardizer.xml`). They are primarily used during the token matching portion of parsing. You can modify these files as needed by adding, deleting, or modifying values in the list. You can also create additional lexicon files.

The PersonName data type includes the following lexicon files:

- `generation.txt`
- `givenNames.txt`
- `salutation.txt`
- `surnames.txt`
- `titles.txt`

These files are located in the `resource` folder under each variant name.

Person Name Normalization Files

Each PersonName variant contains a set of normalization files that are used to normalize input values. The Master Index Standardization Engine uses these files when processing input symbols as defined in the process definition file (`standardizer.xml`). Each normalization file contains a column of unnormalized values, such as nicknames or abbreviations, and a second column that contains the corresponding normalized values. The values in each column are separated by a pipe symbol (|). You can modify these files as needed by adding, deleting, or modifying values in the list. You can also create additional normalization files to reference from the process definition file.

The PersonName data type includes the following normalization files:

- `generationNormalization.txt`
- `givenNameNormalization.txt`
- `salutationNormalization.txt`
- `surnameNormalization.txt`
- `titleNormalization.txt`

These files are located in the `resource` folder under each variant name.

Person Name Process Definition Files

Each variant has its own process definition file (`standardizer.xml`) that defines the state model for standardizing free-form person names. Each of these files also includes a section that defines just normalization without parsing for person names. The process definition file is located in the `resource` folder under each variant name. For information about the structure of this file, see "Process Definition File" on page 17.

Person name standardization has several states, each defining how to process tokens when they are found in certain orders. The default file defines states for salutations, first names, middle names, last names, titles, suffixes, and separators. It defines provisions for instances when the fields do not appear in order or when the input string does not contain complete data. For example, the current definition handles instances where the input string is "FirstName, MiddleName, LastName" as well as instances where the input string is "LastName, FirstName, MiddleName".

The process definition files for person names define several parsing rules for each field component. This file defines a set of cleansing rules to prepare the input string prior to any processing. Then the data is passed to the start state of the FSM. Most fields are preprocessed and then matched against regular expressions or against a list of values in a lexicon file (described in "Person Name Lexicon Files" on page 33). Postprocessing includes replacing regular expressions or normalizing the field value based on a normalization file (described in "Person Name Normalization Files" on page 33). The process definition files also define a set of normalization rules, which are followed when the incoming data already contains name information in separate fields and does not need to be parsed.

Person Name Standardization and Sun Master Index

Master index applications rely on the Master Index Standardization Engine to process person name data. To ensure correct processing of person information, you need to customize the Matching Service for the master index application according to the rules defined for the standardization engine. This includes modifying mefa.xml to define normalization or parsing and phonetic encoding of the appropriate fields. You can modify mefa.xml with the Master Index Configuration Editor in the master index project.

Standardization is defined in the StandardizationConfig section of mefa.xml, which is described in detail in "Match Field Configuration" in *Sun Master Index Configuration Reference* . To configure the required fields for normalization, modify the normalization structure in mefa.xml. To configure the required fields for parsing and normalization, modify the standardization structure. To configure phonetic encoding, modify the phonetic encoding structure. These tasks can all be performed using the Master Index Configuration Editor.

Generally, the person data type processes data that is parsed prior to processing, so you should not need to configure fields to parse unless your person data is stored in free-form text fields with all name information in one field. When processing person data, you might also want to search on address information. In that case, you need to configure the address fields to parse and normalize.

The following topics provide information about the fields used in processing person data and how to configure person data standardization for a master index application. The information provided in these topics is based on the default configuration.

- “Person Name Processing Fields” on page 35
- “Configuring a Normalization Structure for Person Names” on page 36
- “Configuring a Standardization Structure for Person Names” on page 38
- “Configuring Phonetic Encoding for Person Names” on page 39

Person Name Processing Fields

When standardizing person data, not all fields in a record need to be processed by the Master Index Standardization Engine. The standardization engine only needs to process fields that must be parsed, normalized, or phonetically converted. For a master index application, these fields are defined in mefa.xml and processing logic for each field is defined in the standardization engine configuration files.

Person Name Standardized Fields

The Master Index Standardization Engine can process person data that is provided in separate fields within a single record, meaning that no parsing is required of the name fields prior to normalization. It can also process person data contained in one long free-form field and parse the field into its individual components, such as first name, last name, title, and so on. Typically, only first and last names are normalized and phonetically encoded when standardizing person data, but the standardization engine can normalize and phonetically encode any field you choose. By default, the standardization engine processes these fields: first name, middle name, last name, nickname, salutation, generational suffix, and title.

Person Name Object Structure

The fields you specify for person name matching in the Master Index wizard are automatically defined for standardization and phonetic encoding. If you specify the PersonFirstName or PersonLastName match type in the wizard, the following fields are automatically added to the object structure and database creation script:

- *field_name*_Std
- *field_name*_Phon

  where *field_name* is the name of the field for which you specified person name matching.

For example, if you specify the PersonFirstName match type for the FirstName field, two fields, FirstName_Std and FirstName_Phon, are automatically added to the structure. You can also add these fields manually if you do not specify match types in the wizard. If you are parsing free-form person data, be sure all output fields from the standardization process are included in the master index object structure. If you store additional names in the database, such as alias names, maiden names, parent names, and so on, you can modify the phonetic structure to phonetically encode those names as well.

Configuring a Normalization Structure for Person Names

The fields defined for normalization for the PersonName data type can include any name fields. By default, normalization rules are defined in the process definition file for first, middle, and last name fields, and you can easily define additional fields. You only need to define a normalization structure for person data if you are processing individual fields that do not require parsing. Follow the instructions under “Defining Master Index Normalization Rules” in *Sun Master Index Configuration Guide* to define fields for normalization. For the *standardization-type* element, enter **`PersonName`**. For a list of field IDs to use in the *standardized-object-field-id* element, see “Person Name Standardization Components” on page 32.

A sample normalization structure for person data is shown below. This sample specifies that the PersonName standardization type is used to normalize the first name, alias first name, last name, and alias last name fields. For all name fields, both United States and United Kingdom domains are defined for standardization.

```
<structures-to-normalize>
   <group standardization-type="PersonName"
    domain-selector="com.sun.mdm.index.matching.impl.MultiDomainSelector">
      <locale-field-name>Person.PobCountry</locale-field-name>
      <locale-maps>
         <locale-codes>
            <value>UNST</value>
            <locale>US</locale>
         </locale-codes>
         <locale-codes>
            <value>GB</value>
            <locale>UK</locale>
            </locale-codes>
      </locale-maps>
      <unnormalized-source-fields>
         <source-mapping>
            <unnormalized-source-field-name>Person.FirstName
            </unnormalized-source-field-name>
            <standardized-object-field-id>FirstName
            </standardized-object-field-id>
         </source-mapping>
         <source-mapping>
            <unnormalized-source-field-name>Person.LastName
            </unnormalized-source-field-name>
            <standardized-object-field-id>LastName
            </standardized-object-field-id>
         </source-mapping>
      </unnormalized-source-fields>
         <normalization-targets>
            <target-mapping>
               <standardized-object-field-id>FirstName
               </standardized-object-field-id>
               <standardized-target-field-name>Person.FirstName_Std
               </standardized-target-field-name>
            </target-mapping>
            <target-mapping>
               <standardized-object-field-id>LastName
               </standardized-object-field-id>
               <standardized-target-field-name>Person.LastName_Std
               </standardized-target-field-name>
            </target-mapping>
         </normalization-targets>
      </group>
   <group standardization-type="PersonName" domain-selector=
     "com.sun.mdm.index.matching.impl.MultiDomainSelector">
      <locale-field-name>Person.PobCountry</locale-field-name>
      <locale-maps>
         <locale-codes>
            <value>UNST</value>
            <locale>US</locale>
         </locale-codes>
         <locale-codes>
            <value>GB</value>
            <locale>UK</locale>
         </locale-codes>
      </locale-maps>
      <unnormalized-source-fields>
         <source-mapping>
            <unnormalized-source-field-name>Person.Alias[*].FirstName
            </unnormalized-source-field-name>
            <standardized-object-field-id>FirstName
            </standardized-object-field-id>
         </source-mapping>
         <source-mapping>
            <unnormalized-source-field-name>Person.Alias[*].LastName
            </unnormalized-source-field-name>
            <standardized-object-field-id>LastName
            </standardized-object-field-id>
         </source-mapping>
      </unnormalized-source-fields>
      <normalization-targets>
```

```
         <target-mapping>
            <standardized-object-field-id>FirstName
            </standardized-object-field-id>
            <standardized-target-field-name>
            Person.Alias[*].FirstName_Std
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>LastName
            </standardized-object-field-id>
            <standardized-target-field-name>
            Person.Alias[*].LastName_Std
            </standardized-target-field-name>
         </target-mapping>
      </normalization-targets>
   </group>
</structures-to-normalize>
```

Configuring a Standardization Structure for Person Names

For free–form name fields, the source fields that are defined for standardization should include the predefined standardization components. For example, fields containing person name information can include the first name, middle name, last name, suffix, title, and salutation. The target fields you define can include any of these parsed components. Follow the instructions under "Defining Master Index Standardization Rules" in *Sun Master Index Configuration Guide* to define fields for standardization. For the *standardization-type* element, enter **PersonName**. For a list of field IDs to use in the *standardized-object-field-id* element, see "Person Name Standardization Components" on page 32.

A sample standardization structure for person name data is shown below. Only the United States variant is defined in this structure.

```
free-form-texts-to-standardize>
   <group standardization-type="PERSONNAME"
    domain-selector="com.sun.mdm.index.matching.impl.SingleDomainSelectorUS">
      <unstandardized-source-fields>
         <unstandardized-source-field-name>Person.Name
         </unstandardized-source-field-name>
      </unstandardized-source-fields>
      <standardization-targets>
         <target-mapping>
            <standardized-object-field-id>salutation
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Prefix
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>firstName
            </standardized-object-field-id>
            <standardized-target-field-name>Person.FirstName
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>middleName
            </standardized-object-field-id>
            <standardized-target-field-name>Person.MiddleName
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>lastName
            </standardized-object-field-id>
            <standardized-target-field-name>Person.LastName
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>suffix
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Suffix
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>title
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Title
            </standardized-target-field-name>
         </target-mapping>
      </standardization-targets>
   </group>
</free-form-texts-to-standardize>
```

Configuring Phonetic Encoding for Person Names

When you specify a first, middle, or last name field for person name matching in the Master Index wizard, that field is automatically defined for phonetic encoding. You can define additional names, such as maiden names or alias names, for phonetic encoding as well. Follow the instructions under "Defining Phonetic Encoding for the Master Index" in *Sun Master Index Configuration Guide* to define fields for phonetic encoding.

A sample of fields defined for phonetic encoding is shown below. This sample converts name and alias name fields, as well as the street name.

```
<phoneticize-fields>
   <phoneticize-field>
      <unphoneticized-source-field-name>Person.FirstName_Std
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Person.FirstName_Phon
      </phoneticized-target-field-name>
      <encoding-type>Soundex</encoding-type>
   </phoneticize-field>
   <phoneticize-field>
      <unphoneticized-source-field-name>Person.LastName_Std
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Person.LastName_Phon
      </phoneticized-target-field-name>
      <encoding-type>NYSIIS</encoding-type>
   </phoneticize-field>
   <phoneticize-field>
      <unphoneticized-source-field-name>Person.Alias[*].FirstName_Std
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Person.Alias[*].FirstName_Phon
      </phoneticized-target-field-name>
      <encoding-type>Soundex</encoding-type>
   </phoneticize-field>
   <phoneticize-field>
      <unphoneticized-source-field-name>Person.Alias[*].LastName_Std
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Person.Alias[*].LastName_Phon
      </phoneticized-target-field-name>
      <encoding-type>NYSIIS</encoding-type>
   </phoneticize-field>
   <phoneticize-field>
      <unphoneticized-source-field-name>Person.Address[*].AddressLine1_StName
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Person.Address[*].AddressLine1_StPhon
      </phoneticized-target-field-name>
      <encoding-type>NYSIIS</encoding-type>
   </phoneticize-field></phoneticize-fields>
```

FSM–Based Telephone Number Configuration

By default, telephone number data is standardized using the finite state machine (FSM) framework. Processing telephone data involves parsing free-form data fields and normalizing certain field components prior to matching. The following topics describe the default configuration files that define telephone number processing logic and provide information about modifying mefa.xml in a master index application for processing telephone data.

- "Telephone Number Standardization Overview" on page 41
- "Telephone Number Standardization Components" on page 41

- “Telephone Number Standardization Files” on page 42
- “Telephone Number Standardization and Sun Master Index” on page 42

Telephone Number Standardization Overview

Processing data using the PhoneNumber data type includes standardizing and matching telephone numbers. The Master Index Standardization Engine can create the parsed and normalized values for free–form telephone data. These values are required for accurate searching and matching. Several configuration files designed specifically to handle telephone data are included to provide processing logic for the standardization process.

In addition, when processing telephone information, you might want to standardize addresses to enable searching against address information. This requires working with the address configuration files described in Rules–Based Address Data Configuration.

Telephone Number Standardization Components

Standardization engines use tokens to determine how each field is standardized into its individual field components and to determine how to normalize a field value. Tokens also identify the field components to external applications, like a master index application. The following table lists each token generated by the Master Index Standardization Engine for telephone data along with the standardization component they represent. You can only specify the predefined field IDs that are listed in this table unless you customize the existing data type or create a new data type or variant.

TABLE 2 Telephone Number Tokens

Token	Description
areaCode	Represents a field containing an area code.
phoneNumber	Represents a field containing the telephone number, excluding area code, country code, and extension.
extension	Represents a field containing a telephone number extension.
countryCode	Represents a field containing the country code for a telephone number.

Telephone Number Standardization Files

Only one configuration file is used to define standardization logic for processing telephone numbers. The process definition file (`standardizer.xml`) defines the state model and logic for processing telephone numbers. There is only one variant for the PhoneNumber data type that is designed to handle telephone numbers from all countries. The files that make up the variant are stored in the master index project under `PhoneNumber/Generic`. The process definition file is located in the `resource` subdirectory. You can customize this file to fit your processing and standardization requirements for telephone numbers. For more information about the structure of this file, see "Process Definition File" on page 17.

Telephone number standardization has several states, each defining how to process tokens when they are found in certain orders. The default file defines states for country codes, area codes, phone numbers, and extensions. It defines provisions for instances when the fields do not appear in order or when the input string does not contain complete data. For example, the current definition handles instances where the input string begins with a country code or an area code, where it contains an extension, where it does not contain an extension, and when it contains multiple telephone numbers.

The process definition file for telephone numbers define several parsing rules for each field component. This file defines a set of cleansing rules to prepare the input string prior to any processing. Then the data is passed to the start state of the FSM. Most fields are matched against regular expressions and then postprocessed by replacing regular expressions. The output symbols are further processed by concatenating the digit groups of the actual phone number, separated by a hyphen.

Telephone Number Standardization and Sun Master Index

Master index applications rely on the Master Index Standardization Engine to process telephone number data. To ensure correct processing of telephone information, you need to customize the Matching Service for the master index application according to the rules defined for the standardization engine. This includes modifying mefa.xml to define standardization of the appropriate fields. You can modify mefa.xml using the Master Index Configuration Editor.

Standardization is defined in the StandardizationConfig section of mefa.xml, which is described in detail in "Match Field Configuration" in *Sun Master Index Configuration Reference* . To configure the required fields for parsing, modify the standardization structure in mefa.xml.

The following topics provide information about the fields used in processing telephone data and how to configure telephone number standardization for a master index application. The information provided in these topics is based on the default configuration.

- "Telephone Number Processing Fields" on page 43
- "Configuring a Standardization Structure for Telephone Numbers" on page 43

Telephone Number Processing Fields

When standardizing telephone data, not all fields in a record need to be processed by the Master Index Standardization Engine. The standardization engine only needs to process fields that must be parsed, normalized, or phonetically converted. For a master index application, these fields are defined in mefa.xml and processing logic for each field is defined in the Standardization Engine node configuration files.

Telephone Number Standardized Fields

The Master Index Standardization Engine can process telephone data that is contained in one long free-form field and can parse that field into its individual components. By default, the standardization engine separates telephone numbers into these field components: country code, area code, phone number, and extension.

Telephone Number Object Structure

To standardize telephone numbers in a master index application, you need to manually define the standardization structure and you need to add the fields that will store the standardized field components to the object structure. In the default implementation, you can store any combination of the following telephone number field components in the master index database.

- Country Code
- Area Code
- Phone Number
- Extension

The standardization engine has the capability to produce all of the above field components, but you only need to store the ones you need in the master index database.

Configuring a Standardization Structure for Telephone Numbers

For free–form name fields, the source fields you define for standardization should include the standardization components predefined for the PhoneNumber data type. For example, any fields containing telephone number information can include the country code, area code, phone number, and extension. The target fields you define can include any of these parsed fields. Follow the instructions under "Defining Master Index Standardization Rules" in *Sun Master Index Configuration Guide* to define fields for standardization. For the *standardization-type* element, enter **`PhoneNumber`**. For a list of field IDs to use in the *standardized-object-field-id* element, see "Telephone Number Standardization Components" on page 41.

A sample standardization structure for telephone number data is shown below. No variant is defined in this structure because the standardization rules apply to global numbers.

```
<free-form-texts-to-standardize>
   <group standardization-type="PHONENUMBER"
    domain-selector="com.sun.mdm.index.matching.impl.MultiDomainSelector">
      <unstandardized-source-fields>
         <unstandardized-source-field-name>Person.Phone[*].PhoneNumber
         </unstandardized-source-field-name>
      </unstandardized-source-fields>
      <standardization-targets>
         <target-mapping>
            <standardized-object-field-id>countryCode</standardized-object-field-id>
            <standardized-target-field-name>Person.Phone[*].CountryCode
         </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>areaCode</standardized-object-field-id>
            <standardized-target-field-name>Person.Phone[*].AreaCode
         </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>phoneNumber</standardized-object-field-id>
            <standardized-target-field-name>Person.Phone[*].Number
         </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>extension</standardized-object-field-id>
            <standardized-target-field-name>Person.Phone[*].Extension
         </standardized-target-field-name>
         </target-mapping>
      </standardization-targets>
   </group>
</free-form-texts-to-standardize>
```

Rules–Based Address Data Configuration

By default, address standardization is performed using the rules-based framework. Processing street addresses involves parsing, normalizing, and phonetically encoding certain fields prior to matching. The following topics describe the configuration files that define address processing logic and provide instructions for modifying mefa.xml for processing address fields.

- "Address Data Standardization Overview" on page 45
- "Address Data Standardization Components" on page 45
- "Address Data Standardization Files" on page 48
- "Address Standardization and Sun Master Index" on page 57

Address Data Standardization Overview

Processing data using the Address data type includes both standardizing and matching on free-form address fields. The Master Index Standardization Engine can create the parsed, normalized, and phonetic values for address data. These values are needed for accurate searching and matching on address data. You can implement street address standardization and matching on its own, or within an application designed to process person or business information. Standardizing address information allows you to include address fields as search criteria, even though matching might not be performed against these fields.

Several configuration files are designed specifically to handle address data and define processing logic for the standardization and phonetic encoding process. These include address clues files, a patterns file, and a constants file. The United States address standardization engine is based on the work performed at the US Census Bureau. The clues files, in particular, are based on census bureau statistics.

Address Data Standardization Components

Standardization engines use tokens to determine how each field is standardized into its individual field components and to determine how to normalize a field value. Tokens also identify the field components to external applications like a master index application. The following table lists each token generated by the Master Index Standardization Engine for address data along with the standardization component they represent. You can only specify the predefined field tokens that are listed in this table for addresses unless you create a new data type or variant.

TABLE 3 Address Data Tokens

Token	Description
BoxDescript	Represents the PO box type from a standardized address field. By default, this is stored in the *field_name*_StName field in a master index database.
BoxIdentif	Represents the parsed PO box number from a standardized address field. By default, this is stored in the *field_name*_HouseNo field in a master index database.
CenterDescript	Represents the parsed structure description from a standardized address field. This address component is not included in the default master index standardization structure, but you can add it if needed.
CenterIdentif	Represents the parsed structure identifier from a standardized address field. This address component is not included in the default master index standardization structure, but you can add it if needed.

TABLE 3 Address Data Tokens *(Continued)*

Token	Description
ExtraInfo	Represents any extra information that was not included in any of the other parsed components. This address component is not included in the default standardization structure, but you can add it if needed.
HouseClass	Represents the parsed house classification from a standardized address field. This address component is not included in the default master index standardization structure, but you can add it if needed.
HouseNumber	Represents the parsed house number from a standardized address field. By default, this is stored in the *field_name*_HouseNo field in a master index database.
HouseNumPrefix	Represents the parsed house number prefix from a standardized address field (such as the "A" in "A 1587 4th Street"). This address component is not included in the default master index standardization structure, but you can add it if needed.
HouseNumSuffix	Represents the parsed house number suffix from a standardized address field (such as the "B" in "5900 B Arnett Avenue"). This address component is not included in the default master index standardization structure, but you can add it if needed.
MatchPropertyName	Represents the parsed match property name from a standardized address field and is used internally by the standardization engine for blocking and phonetic encoding. This address component is not included in the default master index standardization structure, but you can add it if needed.
MatchStreetName	Represents the parsed and standardized street name from a standardized address field and is used internally by the standardization engine. If you want to store the standardized street name in the database (recommended), map this field to the street name field in the database. By default, this is stored in the *field_name*_StName field in a master index database.
OrigPropertyName	Represents the parsed original property name (such as the name of a complex or business park) from a standardized address field. This address component is not included in the default master index standardization structure, but you can add it if needed.
PropDesPrefDirection	Represents the parsed property direction from a standardized address field. This field ID handles cases where the direction is a prefix to the property description. By default, this is stored in the *field_name*_StDir field in a master index database.
PropDesPrefType	Represents the parsed property type from a standardized address field. This field ID handles cases where the street type is a prefix to the property description. By default, this is stored in the *field_name*_StType field in a master index database.

TABLE 3 Address Data Tokens *(Continued)*

Token	Description
PropertySufDirection	Represents the parsed property direction from a standardized address field. This field ID handles cases where the direction is a suffix to the property description. By default, this is stored in the *field_name*_StDir field in a master index database.
PropertySufType	Represents the parsed property type from a standardized address field. This field ID handles cases where the street type is a suffix to the property description. By default, this is stored in the *field_name*_StType field in a master index database.
RuralRouteDescript	Represents the parsed rural route description from a standardized address field. By default, this is stored in the *field_name*_StName field in a master index database.
RuralRouteIdentif	Represents the parsed rural route identifier from a standardized address field. By default, this is stored in the *field_name*_HouseNo field in a master index database.
SecondHouseNumber	Represents the parsed *second* house number prefix from a standardized address field. This address component is not included in the default master index standardization structure, but you can add it if needed.
SecondHouseNumberPrefix	Represents the parsed *second* house number prefix from a standardized address field (such as “25” in “25 319 10th Ave.”). This address component is not included in the default master index standardization structure, but you can add it if needed.
SecondStreetNameSufDirection	Represents the parsed *second* street direction from a standardized address field. This address component is not included in the default standardization structure, but you can add it if needed.
SecondStreetNameSufType	Represents the parsed *second* street type from a standardized address field. This address component is not included in the default standardization structure, but you can add it if needed.
OrigSecondStreetName	Represents the parsed *second* street name from a standardized address field (for example, an address might include a cross-street or a thoroughfare and dependent thoroughfare). This address component is not included in the default master index standardization structure, but you can add it if needed.
OrigStreetName	Represents the parsed street name from an address field. If you want to store the original street name in the database, map this field to the street name field in the database. This address component is not included in the default standardization structure, but you can add it if needed.

TABLE 3 Address Data Tokens (Continued)

Token	Description
StreetNamePrefDirection	Represents the parsed street direction from a standardized address field. This field ID handles cases where the direction is a prefix to the street name. By default, this is stored in the *field_name*_StDir field in a master index database.
StreetNamePrefType	Represents the parsed street type from a standardized address field. This field ID handles cases where the street type is a prefix to the street name. By default, this is stored in the *field_name*_StType field in a master index database.
StreetNameSufDirection	Represents the parsed street direction from a standardized address field. This field ID handles cases where the direction is a suffix to the street name. By default, this is stored in the *field_name*_StDir field in a master index database.
StreetNameSufType	Represents the parsed street type from a standardized address field. This field ID handles cases where the street type is a suffix to the street name. By default, this is stored in the *field_name*_StType field in a master index database.
StreetNameExtensionIndex	Represents the parsed street name extension from a standardized address field. This address component is not included in the default standardization structure, but you can add it if needed.
WithinStructDescript	Represents the parsed internal descriptor (such as “Floor”) from a standardized address field. This address component is not included in the default standardization structure, but you can add it if needed.
WithinStructIdentif	Represents the parsed internal identifier (such as a floor number) from a standardized address field. This address component is not included in the default standardization structure, but you can add it if needed.

Address Data Standardization Files

Three configuration files define address processing logic for the Master Index Standardization Engine. These files provide information about address patterns and tokens to help the standardization engine determine how to recognize address components and break them out into their respective tokens. You can customize any of the configuration files described in this section to fit your processing and standardization requirements for address data.

The address configuration files are located in the `resource` folder under each variant name for the Address data type. The following topics provide information about each configuration file.

- “Address Clues File” on page 49
- “Address Master Clues File” on page 50
- “Address Patterns File” on page 51
- “Address Pattern File Components” on page 53

Address Clues File

The address clues file (`clues.dat`) lists common terms in street addresses, specifies a normalized value for each common term, and categorizes the terms into street address component types. A term can be categorized into multiple component types. A relevance value specifies which of the component types the term is most likely to be. For example, the term "Junction" is standardized as "Jct" and is classified as a street type, building unit, and generic term (giving relevance in that order).

This file helps the Master Index Standardization Engine recognize common terms in street addresses in order to parse and normalize the values correctly. The syntax of this file is:

```
common-term normalized-term ID-number/type-token
```

You can modify or add entries in this table as needed. The following table describes the columns in the address clues file.

TABLE 4 Address Clues File Columns

Column	Description
common-term	A term commonly found in street addresses.
normalized-term	The normalized version of the common term.
ID-number/type-token	An ID number and a token indicating the type of address component represented by the common term. The ID number corresponds to an ID number in the address master clues file, and the type token corresponds to the type specified for that ID number in the address master clues file. One term might have several ID number and token type pairs. Their order of appearance indicates their relevance value.

Following is an excerpt from the US address clues file.

```
TRLR VLG          Trpk            59BU
TRPK              Trpk            59BU
TRPRK             Trpk            59BU
VILLA             Vlla            305TY          60BU
VLLA              Vlla            305TY          60BU
VILLAS            Vlla            60BU
VILL              Vlg             317TY          61BU         364AU
VILLAG            Vlg             317TY          61BU         364AU
VLG               Vlg             317TY          61BU         364AU
VILLAGE           Vlg             317TY          61BU         364AU
VILLG             Vlg             317TY          61BU         364AU
VILLIAGE          Vlg             317TY          61BU         364AU
VLGE              Vlg             317TY          61BU         364AU
VIVI              Vivi            62BU
```

```
VIVIENDA          Vivi            62BU
COLLEGE           Coll            64BU                       0AU
CLG               Coll            64BU
COTTAGE           Cott            65BU           65BP        0AU
```

Address Master Clues File

The address master clues file (`masterClues.dat`) lists common terms in street addresses as defined by the United States Postal Service (USPS), the United Kingdom’s Royal Mail, the Australian Postal Corporation, or France’s La Poste (depending on the variant in use). For each common term, this file specifies a normalized value, defines postal information, and categorizes the terms into street address component types. A term can be categorized into multiple component types.

The syntax of this file is:

```
ID-number common-term normalized-term short-abbrev postal-abbrev CFCCS
type-token usage-flag postal-flag
```

You can modify or add entries in this table as needed. The following table describes the columns in the address master clues file.

TABLE 5 Address Master Clue File Columns

Column	Description
ID-number	A unique identification number for the address common term. This number corresponds to an ID number for the same term in the address clues file.
common-term	A common address term, such as Park, Village, North, Route, Centre, and so on.
normalized-term	The normalized version of the common term.
short-abbrev	A short abbreviation of the common term.
postal-abbrev	The standard postal abbreviation of the common term.
CFCCS	The census feature class code of the term (as defined in the Census Tiger® database). The following values are used: ■ **A** – Road ■ **B** – Railroad ■ **C** – Miscellaneous ■ **D** – Landmark ■ **E** – Physical feature ■ **F** – Nonvisible feature ■ **H** – Hydrography ■ **X** – Unclassified

TABLE 5 Address Master Clue File Columns *(Continued)*

Column	Description
type-token	The type of address component represented by the common term. Types are specified by an address token (for more information, see "Address Type Tokens" on page 53).
usage-flag	A flag indicating how the term is used (for more information, see "Pattern Classes" on page 56)
postal-flag	The standard postal code for the term.

Following is an excerpt from the US address master clues file.

```
11Alley                    Alley            Al         Aly A        TY R U
12Alternate Route          Alt Rte          Alt        Alt A        TY R
15Arcade                   Arcade           Arc        Arc A        TY R U
16Arroyo                   Arroyo           Arryo      ArryHA       TY R
17Autopista                Atpta            Apta       AptaA        TY R
18Avenida                  Avenida          Ava        Ava A        TY R
19Avenue                   Avenue           Ave        Ave A        TY R U
26Boulevard                Blvd             Blvd       BlvdA        TY R U
32Bulevar                  Blvr             Blv        Blv A        TY R
33Business Route           Bus Rte          BusRt      BsRtA        TY R
34Bypass                   Bypass           Byp        Byp A        TY R U
36Calle                    Calle            Calle      ClleA        TY R
37Calleja                  Calleja          Cja        Cja A        TY R
38Callejon                 Callej           Cjon       CjonA        TY R
39Camino                   Camino           Cam        Cam A        TY R
47Carretera                Carrt            Carr       CarrA        TY R
48Causeway                 Cswy             Cswy       CswyAH       TY R U
51Center                   Center           Ctr        Ctr DA       TY R U
```

Address Patterns File

The address patterns file (`patterns.dat`) defines the expected input patterns of each individual street address field being standardized so the Master Index Standardization Engine can recognize and process these values. Tokens indicate the type of address component in the input and output fields. This file contains two rows for each pattern. The first row defines the input pattern for each address field and provides an example. The second row defines the output pattern for each address field, the pattern type, the relative importance of the pattern compared to other patterns, and usage flags. Below is an example.

```
AU A1 TY                01 Oak B Street
NA NA ST                T* 75                TX
```

When an address is parsed, each line of the address is delineated by a pipe (|) and sent to the parser separately. The output tokens for each line are then concatenated and the output pattern is processed using the address patterns file to determine whether the output pattern is listed in the file. If the pattern is found, output patterns are modified as indicated in the patterns file to resolve any ambiguities that might arise when two lines of address information contain common elements. The relative importance determines which pattern to use when the format of the input field matches more than one pattern. This file should only be modified by personnel with a thorough understanding of address patterns and tokens.

The syntax of this file is:

```
input-pattern example output-pattern pattern-class pattern-modifier priority
usage-flag exclude-flag
```

You can modify or add entries in this table as needed. The following table describes the columns in the address patterns file.

TABLE 6 Address Patterns File

Column	Description
input-pattern	Tokens that represent a possible input pattern from an individual unparsed street address field. Each token represents one component. For more information about address tokens, see “Address Type Tokens” on page 53.
example	An example of a street address that fits the specified pattern. This file element is optional.
output-pattern	Tokens that represent the output pattern for the specified input pattern. Each token represents one component of the output of the Master Index Standardization Engine. For more information about address tokens, see “Address Type Tokens” on page 53.
pattern-class	An indicator of the type of address component represented by the pattern. Possible pattern types are listed in “Pattern Classes” on page 56“Pattern Classes” on page 56.
pattern-modifier	An indicator of whether the priority of the pattern is averaged against other patterns that match the input. Pattern modifiers are listed in “Pattern Modifiers” on page 56.
priority	The priority weight to use for the pattern when the pattern is a sub-pattern of a larger input pattern. For more information, see “Priority Indicators” on page 56.
usage-flag	A flag indicating how the term is used (for more information, see “Pattern Classes” on page 56). This file element is optional.
exclude-flag	This file element is optional.

Following is an excerpt from the address patterns file.

```
NU DR TY A1 AU                     01   123 South Avenida B Oak
HN PD PT NA NA                     H* 70

NU DR TY NU DR                     01   123 South Avenida 1 West
HN PD PT NA SD                     H* 70

NU A1 TY AU TY                     01   123 C circle hill drive
HN HS NA NA ST                     H* 70

NU A1 AM A1 TY                     01   123 M & M road
HN NA NA NA ST                     H* 65

NU TY AU A1                        01   123 Avenida Oak B
HN PT NA NA                        H* 60

NU TY NU A1                        01   123 Avenida 1 B
HN PT NA NA                        H* 60
```

Address Pattern File Components

The address patterns files use pattern type tokens, pattern classes, pattern modifiers, and priority indicators to process and parse address data. Before modifying any of the patterns files, you must have a good understanding of these file components.

Address Type Tokens

The address pattern and clues files use tokens to denote different components in a street address, such as street type, house number, street names, and so on. These files use one set of tokens for input fields and another set for output fields. You can use only the predefined tokens to represent address components; the Master Index Standardization Engine does not recognize custom tokens.

The following table lists and describes each input token.

TABLE 7 Input Address Pattern Type Tokens

Token	Description
A1	Alphabetic value, one character in length
AM	Ampersand
AU	Generic word
BP	Building property
BU	Building unit

TABLE 7 Input Address Pattern Type Tokens *(Continued)*

Token	Description
BX	Post office box
DA	Dash (as a starting character)
DR	Street direction
EI	Extra information
EX	Extension
FC	Numeric fraction
HR	Highway route
MP	Mile posts
NL	Common words, such as "of", "the", and so on
NU	Numeric value
OT	Ordinal type
PT	Prefix type
RR	Rural route
SA	State abbreviation
TY	Street type
WD	Descriptor within the structure
WI	Identifier within the structure

The following table lists and describes each output token.

TABLE 8 Output Address Pattern Tokens

Token	Description
1P	Building number prefix
2P	Second building number prefix
BD	Property or building directional suffix
BI	Structure (building) identifier
BN	Property or building name
BS	Building number suffix

TABLE 8 Output Address Pattern Tokens *(Continued)*

Token	Description
BT	Property or building type suffix
BX	Post office box descriptor
BY	Structure (building) descriptor
DB	Property or building directional prefix
EI	Extra information
EX	Extension index
H1	First house number (the actual number)
H2	Second house number (house number suffix)
HN	House number
HS	House number suffix
N2	Second street name
NA	Street name
NB	Building number
NL	Conjunctions that connect words or phrases in one component type (usually the street name)
P1	House number prefix
P2	Second house number prefix
PD	Directional prefix to the street name
PT	Street type prefix to the street name
RR	Rural route descriptor
RN	Rural route identifier
S2	Street type suffix to the second street name
SD	Directional suffix to the street name
ST	Street type suffix to the street name
TB	Property or building type prefix
WI	Identifier within the structure
WD	Descriptor within the structure
XN	Post office box identifier

Pattern Classes

Each pattern defined in the address patterns file must have an associated pattern class. The pattern class indicates a portion of the input pattern or the type of address data that is represented by the pattern. You can specify any of the following pattern classes.

- **H** - the address pattern represents a house
- **B** - the address pattern represents a building
- **W** - the address pattern represents a unit within a structure, such as an apartment or suite number
- **T** - the address pattern represents a street type or direction
- **R** - the address pattern represents a rural route
- **P** - the address pattern represents a Post Office box
- **N** - the address pattern is mostly numeric

These classes are also specified as usage flags in the patterns file and the master clues file.

Pattern Modifiers

Each pattern type must be followed by a pattern modifier that indicates how to handle cases where one or more defined patterns is found to be a sub-pattern of a larger input pattern. In this case, the Master Index Standardization Engine must know how to prioritize each defined pattern that is a part of the larger pattern. There are two pattern modifiers.

- * - An asterisk indicates that the priority weight for the matching pattern is averaged down equally with the other matching sub-patterns.
- + - A plus sign indicates that the priority weight for the matching pattern is not averaged down equally with the other matching sub-patterns.

Priority Indicators

The priority indicator is a numeric value following the pattern modifier that indicates the priority weight of the pattern. These values work best when defined as a multiple of five between and including 35 and 95. If a pattern is assigned a priority of 90 or 95 and the pattern matches, or is a sub-pattern of, the input pattern, the standardization engine stops searching for additional matching patterns and uses the high-priority matching pattern.

Address Standardization and Sun Master Index

Master index applications rely on the Master Index Standardization Engine to process address data. To ensure correct processing of address information, you need to customize the Matching Service for the master index application according to the rules defines for the standardization engine. This includes modifying mefa.xml to define parsing and phonetic encoding of the appropriate fields. You can use the Master Index Configuration Editor to modify mefa.xml.

Standardization is defined in the StandardizationConfig section of mefa.xml, which is described in detail in "Match Field Configuration" in *Sun Master Index Configuration Reference* . To configure the required fields for parsing and normalization, modify the standardization structure in mefa.xml. To configure phonetic encoding, modify the phonetic encoding structure. You can perform all of these tasks using the Master Index Configuration Editor.

Generally, the address data type processes data that requires parsing prior to processing. You should not need to configure fields to normalize for addresses. The following topics provide information about the fields used in processing address data and how to configure address data standardization for a master index application. The information provided in these topics is based on the default configuration.

- "Address Data Processing Fields" on page 57
- "Configuring a Standardization Structure for Address Data" on page 58
- "Configuring Phonetic Encoding for Address Data" on page 60

Address Data Processing Fields

When standardizing address data, not all fields in a record need to be processed by the Master Index Standardization Engine. The standardization engine only needs to process address fields that must be parsed, normalized, or phonetically converted. For a master index application, these fields are defined in mefa.xml and processing logic for each field is defined in the Standardization Engine node configuration files.

Address Standardized Fields

The Master Index Standardization Engine expects that street address data will be provided in a free-form text field containing several components that must be parsed. By default, the standardization engine is configured to parse these components and to normalize and phonetically encode the street name. You can specify additional fields for phonetic encoding.

If you specify the Address match type for any field in the wizard, a standardization structure for that field is defined in mefa.xml. The fields listed under "Address Object Structure" on page 58 are automatically defined as the target fields. Each of these fields has several entries in the standardization structure. This is because different parsed components can be stored in the same field. For example, the house number, post office box number, and rural route identifier are all stored in the house number field. If you do not specify address fields for matching in the wizard but want to standardize the fields, you can create a standardization structure in mefa.xml using the Master Index Configuration Editor.

Address Object Structure

The address fields specified for standardization are parsed into several additional fields. If you specify the Address match type in the wizard, the following fields are automatically added to the object structure and database creation script.

- *field_name*_HouseNo
- *field_name*_StName
- *field_name*_StDir
- *field_name*_StType
- *field_name*_StPhon

  where *field_name* is the name of the field for which you specified address matching. For example, if you specify the Address match type for the AddressLine1 field, the following fields are automatically added to the structure: AddressLine1_HouseNo, AddressLine1_StName, AddressLine1_StDir, AddressLine1_StType, and AddressLine1_StPhon.

You can add these fields manually if you do not specify a match type in the wizard.

Configuring a Standardization Structure for Address Data

For free–form address fields, the source fields you define for parsing should include the standardization components that are predefined for parsing and normalization. For example, fields containing address information can include any of the field components listed in "Address Data Standardization Components" on page 45. The target fields can include any of these parsed fields. Follow the instructions under "Defining Master Index Standardization Rules" in *Sun Master Index Configuration Guide* to define fields for standardization. For the *standardization-type* element, enter **Address**. For a list of field IDs to use in the *standardized-object-field-id* element, see "Address Data Standardization Components" on page 45.

Note – In the default configuration, the rules defined for the address data type assume that all input fields must be parsed as well as normalized. Thus, there is no need to configure fields only for normalization.

A sample standardization structure for address data is shown below. This structure parses the first two lines of street address into the standard street address fields. Only the United States variant is defined in this structure.

```
free-form-texts-to-standardize>
   <group standardization-type="ADDRESS"
    domain-selector="com.sun.mdm.index.matching.impl.SingleDomainSelectorUS">
      <unstandardized-source-fields>
         <unstandardized-source-field-name>Person.Address[*].Address1
         </unstandardized-source-field-name>
         <unstandardized-source-field-name>Person.Address[*].Address2
         </unstandardized-source-field-name>
      </unstandardized-source-fields>
      <standardization-targets>
         <target-mapping>
            <standardized-object-field-id>HouseNumber
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Address[*].HouseNumber
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>RuralRouteIdentif
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Address[*].HouseNumber
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>BoxIdentif
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Address[*].HouseNumber
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>MatchStreetName
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Address[*].StreetName
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>RuralRouteDescript
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Address[*].StreetName
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>BoxDescript
            </standardized-object-field-id>
            <standardized-target-field-name>Person.Address[*].StreetName
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>PropDesPrefDirection
```

```
                </standardized-object-field-id>
                <standardized-target-field-name>Person.Address[*].StreetDir
                </standardized-target-field-name>
             </target-mapping>
             <target-mapping>
                <standardized-object-field-id>PropDesSufDirection
                </standardized-object-field-id>
                <standardized-target-field-name>Person.Address[*].StreetDir
                </standardized-target-field-name>
             </target-mapping>
             <target-mapping>
                <standardized-object-field-id>StreetNameSufType
                </standardized-object-field-id>
                <standardized-target-field-name>Person.Address[*].StreetType
                </standardized-target-field-name>
             </target-mapping>
             <target-mapping>
                <standardized-object-field-id>StreetNamePrefType
                </standardized-object-field-id>
                <standardized-target-field-name>Person.Address[*].StreetType
                </standardized-target-field-name>
             </target-mapping>
          </standardization-targets>
       </group>
    </free-form-texts-to-standardize>
```

Configuring Phonetic Encoding for Address Data

When you match or standardize on street address fields, the street name should be specified for phonetic conversion (this is done by default in a master index application). Follow the instructions under "Defining Phonetic Encoding for the Master Index" in *Sun Master Index Configuration Guide* to define fields for phonetic encoding.

A sample of the *phoneticize-fields* element is shown below. This sample only converts the address street name. You can define additional fields for phonetic encoding.

```
<phoneticize-fields>
   <phoneticize-field>
      <unphoneticized-source-field-name>Person.Address[*].StreetName
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Person.Address[*].StreetName_Phon
      </phoneticized-target-field-name>
      <encoding-type>NYSIIS</encoding-type>
   </phoneticize-field>
</phoneticize-fields>
```

Rules-Based Business Name Configuration

By default, business name standardization is performed using the rules-based framework. Processing business name fields involves parsing, normalizing, and phonetically encoding certain fields prior to matching. The following topics describe the configuration files that define business name processing logic and provide instructions for modifying mefa.xml for processing business names.

- "Business Name Standardization Overview" on page 61
- "Business Name Standardization Components" on page 61
- "Business Name Standardization Files" on page 62
- "Business Name Standardization and Sun Master Index" on page 74

Business Name Standardization Overview

Processing data using the BusinessName data type includes both standardizing and matching on free-form business name fields. The Master Index Standardization Engine can create the parsed, normalized, and phonetic values for business names. These values are needed for accurate searching and matching on business information. You can implement business name standardization and matching on its own, or within an application designed to process person information. Standardizing business name fields allows you to include these fields as search criteria, even though matching might not be performed against these fields.

The Master Index Standardization Engine can create standardized and phonetic values for business name field components. Several configuration files are designed specifically to handle business names to define additional logic for the standardization and phonetic encoding process. These include reference files, a patterns file, and key type files. The business name standardization files are contained in one generic variant.

Business Name Standardization Components

Standardization engines use tokens to determine how each field is standardized into its individual field components and to determine how to normalize a field value. Tokens also identify the field components to external applications like a master index application. The following table lists each token generated by the Master Index Standardization Engine for business names along with the standardization component they represent. You can only specify the predefined field tokens that are listed in this table for business names unless you create a new data type or variant.

TABLE 9 Business Name Tokens

Token	Description
PrimaryName	Represents the name parsed from a free-form text business name field.
OrgTypeKeyword	Represents the organization type parsed from a free-form text business name field.
AssocTypeKeyword	Represents the association type parsed from a free-form text business name field.
IndustrySectorList	Represents the industry sector parsed a free-form text business name field.
IndustryTypeKeyword	Represents the industry type parsed from a free-form text business name field (industry type is a subset of the sector).
AliasList	Represents the alias parsed from a free-form text business name field.
Url	Represents the URL parsed from a free-form text business name field.

Business Name Standardization Files

Several configuration files are used to define business name processing logic for the Master Index Standardization Engine. These files provide information about business name patterns and tokens to help the standardization engine determine how to recognize business name components and break them out into their respective tokens. You can customize any of the configuration files described in this section to fit your processing and standardization requirements for business names.

The following topics described each file used for business name standardization:

- "Business Name Adjectives Key Type File" on page 63
- "Business Alias Key Type File" on page 63
- "Business Association Key Type File" on page 64
- "Business General Terms Reference File" on page 64
- "Business City or State Key Type File" on page 65
- "Business Former Name Reference File" on page 66
- "Merged Business Name Category File" on page 66
- "Primary Business Name Reference File" on page 67
- "Business Connector Tokens Reference File" on page 68
- "Business Country Key Type File" on page 68
- "Business Industry Sector Reference File" on page 69
- "Business Industry Key Type File" on page 70
- "Business Organization Key Type File" on page 71
- "Business Patterns File" on page 72

Business Name Adjectives Key Type File

The adjectives key type file (`bizAdjectivesTypeKeys.dat`) defines adjectives commonly found in business names so the Master Index Standardization Engine can recognize and process these values as a part of the business name. This file contains one column with a list of commonly used adjectives, such as General, Financial, Central, and so on.

You can modify or add entries in this file as needed. Following is an excerpt from the adjectives key type file.

```
DIGITAL
DIRECTED
DIVERSIFIED
EDUCATIONAL
ELECTROCHEMICAL
ENGINEERED
EVOLUTIONARY
EXTENDED
FACTUAL
FEDERAL
```

Business Alias Key Type File

The alias key type file (`bizAliasTypeKeys.dat`) lists business name acronyms and abbreviations along with their standardized names so the standardization engine can recognize and process these values correctly. You can add entries to the alias key type file using the following syntax.

```
alias standardized-name
```

The following table describes the columns in the alias key type file.

TABLE 10 Alias Key Type File

Column	Description
alias	An abbreviation or acronym commonly used in place of a specific business name.
standardized-name	The normalized version of the alias name.

Following is an excerpt from the alias key type file.

```
BBH                 BARTLE BOGLE HEGARTY
BBH                 BROWN BROTHERS HARRIMAN
IBM                 INTERNATIONAL BUSINESS MACHINE
IDS                 INCOMES DATA SERVICES
IDS                 INSURANCE DATA SERVICES
IDS                 THE INTEGRATED DECISION SUPPORT GROUP
IDS                 THE INTERNET DATABASE SERVICE
CAL-TECH            CALIFORNIA INSTITUTE OF TECHNOLOGY
```

Business Association Key Type File

The association key type file (`bizAssociationTypeKeys.dat`) lists business association types along with their standardized names so the standardization engine can recognize and process these values correctly. You can add entries to the association key type file using the following syntax.

```
association-type standardized-type
```

The following table describes the columns in the association key type file.

TABLE 11 Association Key Type Table

Column	Description
association-type	A common association type for businesses, such as Partners, Group, and so on.
standardized-type	The standardized version of the association type. If this column contains a name instead of a zero, that name must also be listed in a different entry as an association type with a standardized form of “0”.

Following is an excerpt from the `bizAssociationTypeKeys.dat` file.

```
ASSOCIATES          0
BANCORP             0
BANCORPORATION      BANCORP
COMPANIES           0
GP                  GROUP
GROUP               0
PARTNERS            0
```

Business General Terms Reference File

The general terms reference file (`bizBusinessGeneralTerms.dat`) lists terms commonly used in business names. This file is used to identify terms that indicate a business, such as bank, supply, factory, and so on, so the Master Index Standardization Engine can recognize and process the business name.

This file contains one column that lists common terms in the business names you process. You can add entries as needed. Below is an excerpt from the general terms reference file.

```
BUILDING
CITY
CONSUMER
EAST
EYE
FACTORY
LATIN
NORTH
SOUTH
```

Business City or State Key Type File

The city or state key type file (`bizCityorStateTypeKeys.dat`) lists various cities and states that might be used in business names. It also classifies each entry as a city (CT) or state (ST) and indicates the country in which the city or state is located. This enables the standardization engine to recognize and process these values correctly. You can add entries to the city or state key type file using the following syntax.

```
city-or-state type country
```

The following table describes the columns in the file.

TABLE 12 City or State Key Type File

Column	Description
city-or-state	The name of a city or state used in business names.
type	An indicator of whether the value is a city or state. “CT” indicates city and “ST” indicates state.
country	The country code of the country in which the city or state is located.

Following is an excerpt from the city or state key type file.

```
ADELAIDE                 CT   AU
ALABAMA                  ST   US
ALASKA                   ST   US
ALGIERS                  CT   DZ
AMSTERDAM                CT   NL
ARIZONA                  ST   US
ARKANSAS                 ST   US
ASUNCION                 CT   PY
ATHENS                   CT   GR
```

Business Former Name Reference File

The business former name reference file (`bizCompanyFormerNames.dat`) provides a list of common company names along with names by which the companies were formerly known so the standardization engine can recognize a business when processing a record containing a previous business name. You can add entries to the business former name table using the following syntax.

```
former-name current-name
```

The following table describes each column in the business former name reference file.

TABLE 13 Business Former Name Reference File

Column	Description
former-name	One of the company's previous names.
current-name	The company's current name.

Below is an excerpt from the business former name reference file.

```
HELLENIC BOTTLING                       COCA-COLA HBC
INTERNATIONAL PRODUCTS                  THE TERLATO WINE
ORGANIC FOOD PRODUCTS                   SPECTRUM ORGANIC PRODUCTS
SUTTER HOME WINERY                      TRINCHERO FAMILY ESTATES
```

Merged Business Name Category File

The merged business name category file (`bizCompanyMergerNames.dat`) provides a list of companies whose name changed because of a merger along with the name of the company after the merge. It also classifies the business names into industry sectors and sub-sectors. This enables the standardization engine to recognize the current company name and determine the sector of the business. You can add entries to the business merger name file using the following syntax.

```
former-name/merged-name sector-code
```

The following table describes each column in the merged business name category file.

TABLE 14 Merged Business Name Category File

Column	Description
former-name	The name of the company whose name was not kept after the merger.

TABLE 14 Merged Business Name Category File *(Continued)*

Column	Description
merged-name	The name of the company whose name was kept after the merger.
sector-code	The industry sector code of the business. Sector codes are listed in the `bizIndustryCategoriesCode.dat` file.

Below is an excerpt from the merged business name category file.

```
DUKE/FLUOR DANIEL                                 20005
FAULTLESS STARCH/BON AMI                          09004
FIND/SVP                                          10013
FIRST WAVE/NEWPARK SHIPBUILDING                   27005
GUNDLE/SLT                                        19020
HMG/COURTLAND                                     23004
J BROWN/LMC                                       10014
KORN/FERRY                                        10020
LINSCO/PRIVATE LEDGER                             14005
```

Primary Business Name Reference File

The primary business name reference file (`bizCompanyPrimaryNames.dat`) provides a list of companies by their primary name. It also classifies the business names into industry sectors and sub-sectors. This enables the standardization engine to determine the correct value of the sector field when parsing the business name. You can add entries to the primary business name file using the following syntax.

```
primary-name sector-code
```

The following table describes the columns in the primary business name reference file.

TABLE 15 Primary Business Name Reference File

Column	Description
primary-name	The primary name of the company.
sector-code	The industry sector code of the business. Sector codes are listed in the `bizIndustryCategoriesCode.dat` file.

Below is an excerpt from the primary business name reference file.

```
BROTHER INTERNATIONAL                             12006
BRYSTOL-MYERS SQUIBB                              11005
BURLINGTON COAT FACTORY                           24003
BURLINGTON NORTHERN SANTA FE                      27005
BV SOLUTIONS                                      06012
CABLEVISION                                       26001
CABOT                                             04006
CADENCE                                           06010
CAMPBELL                                          22006
CAPITAL BLUE CROSS                                17001
```

Business Connector Tokens Reference File

The connector tokens reference file (`bizConnectorTokens.dat`) defines common values (typically conjunctions) that connect words in business names. For example, in the business name "Nursery of Venice", "of" is a connector token. This helps the standardization engine recognize and process the full name of a business by indicating that the token connects two parts of the full name.

This file contains one column that lists the connector tokens in the business names you process. You can add entries as needed. Below is an excerpt from the connector tokens reference file.

```
AN
DE
DES
DOS
LA
LAS
LE
OF
THE
```

Business Country Key Type File

The country key type file (`bizCountryTypeKeys.dat`) lists countries and continents, along with their abbreviations and assigned nationalities. For continents, the abbreviation is "CON" to separate them from countries. This enables the standardization engine to recognize and process these values as countries or continents. You can add entries to the country key type file using the following syntax.

```
country abbreviation nationality
```

The following table describes the columns in the country key type file.

TABLE 16 Country Key Type File

Column	Description
country	The name of a country or continent.
abbreviation	The common abbreviation for the specified country. The abbreviation for a continent is always “CON”.
nationality	The nationality assigned to a person or business originating in the specified country.

Following is an excerpt from the country key type file.

```
AMERICA                        CON  AMERICAN
AFRICA                         CON  AFRICAN
EUROPE                         CON  EUROPEAN
ASIA                           CON  ASIAN
AFGHANISTAN                    AF   AFGHAN
ALBANIA                        AL   ALBANIAN
ALGERIA                        DZ   ALGERIAN
```

Business Industry Sector Reference File

The industry sector reference file (`bizIndustryCategoryCode.dat`) lists and groups various industry sectors and sub-sectors, and includes an identification code for each type so the standardization engine can identify and process the industry sectors for different businesses. You can add entries to the industry sector reference file using the following syntax.

```
sector-code industry-sector
```

The following table describes each column in the industry sector reference file.

TABLE 17 Industry Sector Reference File

Column	Description
sector-code	The identification code of the specified sector. The first two numbers of each code identify the general industry sector; the last three number identify a sub-sector.
industry-sector	A description of the industry category. This is written in the format “`sector - sub-sector`”, where `sector` is a general category of industry types, and `sub-sector` is a specific industry within that category.

Following is an excerpt from the industry sector reference file.

```
02006        Automotive & Transport Equipment - Recreational Vehicles
02007        Automotive & Transport Equipment - Shipbuilding & Related Services
02008        Automotive & Transport Equipment - Trucks, Buses & Other Vehicles
03001        Banking - Banking
04001        Chemicals - Agricultural Chemicals
04002        Chemicals - Basic & Intermediate Chemicals & Petrochemicals
04003        Chemicals - Diversified Chemicals
04004        Chemicals - Paints, Coatings & Other Finishing Products
04005        Chemicals - Plastics & Fibers
04006        Chemicals - Specialty Chemicals
05001        Computer Hardware - Computer Peripherals
05002        Computer Hardware - Data Storage Devices
05003        Computer Hardware - Diversified Computer Products
```

Business Industry Key Type File

The industry key type file (`bizIndustryTypeKeys.dat`) is used to standardize the value of the Industry field into common industries to which businesses belong so the standardization engine can recognize and process the industry types for different businesses. You can add entries to the industry key type file using the following syntax.

```
industry-type standardized-form sectors
```

The following table describes each column in the industry key type file.

TABLE 18 Industry Key Type File

Column	Description
industry-type	The original value of the industry type in the input record.
standardized-form	The normalized version of the industry type. If this column contains a name instead of a zero, that name must also be listed in a different entry as an industry type with a standardized form of "0".
sectors	The industry categories of the specified industry type. These values correspond to the sector codes listed in the industry sector file (`bizIndustryCategoryCode.dat`). You can list as many categories as apply for each type, but they must be entered with a space between each and no line breaks, and they must correspond to an entry in the industry sector file.

Below is an excerpt from the industry key type file.

```
TECH              TECHNOLOGY        05001-05007
TECHNOLOGIES      TECHNOLOGY        05001-05007
TECHNOLOGY        0                 05001-05007
TECHSYSTEMS       0                 05001-05007
```

```
TELE PHONE         TELEPHONE          16005
TELE PHONES        TELEPHONES         16005
TELEVISION         TV                 11013  21014
TELECOM            0                  16005  26006  26009  26010
TELECOMM           TELECOMMUNICATION  16005  26006  26008
TELECOMMUNICATION  0                  16005  26006  26008
```

Business Organization Key Type File

The organization key type file (`bizOrganizationTypeKeys.dat`) is used to standardize the value of the Organization field into common organizations to which businesses belong. This helps the standardization engine recognize and process the organization types for different businesses. You can add entries to the organization key type file using the following syntax.

```
original-type standardized-form
```

The following table describes each column in the organization key type file.

TABLE 19 Organization Key Type File

Column	Description
original-type	The original value of the organization field in an input record.
standardized-form	The normalized version of an organization type. A zero (0) in this field indicates that the value in the first column is already in its standardized form. If this column contains a name instead of a zero, that name must also be listed in a different entry as an original type with a standardized form of "0".

Below is an excerpt from the organization key type file.

```
INC                INCORPORATED
INCORPORATED       0
KG                 0
KK                 0
LIMITED            0
LIMITED PARTNERSHIP 0
LLC                0
LLP                0
LP                 LIMITED PARTNERSHIP
LTD                LIMITED
```

Business Patterns File

The business patterns file (`bizpatterns.dat`) defines multiple formats expected from the business name input fields along with the standardized output of each format. The patterns and output appear in two-row pairs in this file, as shown below.

```
4 PNT AST SEP-GLC ORT
PNT AST DEL ORT
```

The first line describes the input pattern and the second describes the output pattern using tokens to denote each component. The supported tokens are described in "Business Name Tokens" on page 73. A number at the beginning of the first line indicates the number of components in the given business name format. You can modify this file using the following syntax.

```
length input-pattern
output-pattern
```

The following table lists and describes the components in the above syntax.

TABLE 20 Business Patterns File Components

Component	Description
length	The number of business name components in the input field.
input-pattern	Tokens that represent a possible input pattern from the unparsed business name fields. Each token represents one component. For more information about address tokens, see "Business Name Tokens" on page 73.
output-pattern	Tokens that represent the output pattern for the specified input pattern. Each token represents one component. For more information about business name tokens, see "Business Name Tokens" on page 73.

Below is an excerpt from the business patterns file.

```
4 PNT AST SEP-GLC ORT
PNT AST DEL ORT

4 NFG AJT SEP-GLC ORT
PNT PNT DEL ORT

4 NF AJT SEP-GLC ORT
PNT PNT DEL ORT

4 CST IDT NF ORT
PNT PNT PNT ORT
```

```
4 PNT AJT SEP-GLC ORT
PNT PNT DEL ORT
```

Business Name Tokens

The business patterns file uses tokens to denote different components in a business name, such as the primary name, alias type key, URL, and so on. The file uses one set of tokens for input fields and another set for output fields. The tokens indicate the type key files to use to determine the appropriate values for each output field. You can use only the predefined tokens to represent business name components; the standardization engine does not recognize custom tokens.

Table 21 lists and describes each input token; Table 22 lists and describes each output token.

TABLE 21 Business Name Input Pattern Tokens

Pattern Identifier	Description
CTT	A connector token
PNT	A primary name of a business
PN-PN	A hyphenated primary name of a business
BCT	A common business term
URL	The URL of the business' web site
ALT	A business alias type key (usually an acronym)
CNT	A country name
NAT	A nationality
CST	A city or state type key
IDT	An industry type key
IDT-AJT	Both an industry and an adjective type key
AJT	An adjective type key
AST	An association type key
ORT	An organization type key
SEP	A separator key
NFG	Generic term, not recognized as a specific business name component, with an internal hyphen
NF	Generic term, not recognized as a specific business name component

TABLE 21 Business Name Input Pattern Tokens *(Continued)*

Pattern Identifier	Description
NFC	A single character, not recognized as a specific business name component
SEP-GLC	A joining comma (a *glue type separator*)
SEP-GLD	A joining hyphen (a *glue type separator*)
AND	The text "and"
GLU	A glue type key, such as a forward slash, connecting two parts of a business name component
PN-NF	A business primary name followed by a hyphen and a generic term that is not recognized as a specific business name component
NF-PN	A generic term that is not recognized as a specific business name component, followed by a hyphen and a recognized business primary name
NF-NF	Two generic terms, not recognized as specific business name components and separated by a hyphen

Table 22 lists and describes each output token.

TABLE 22 Business Name Output Pattern Tokens

Pattern Identifier	Description
PNT	The primary name of the business
URL	The URL of the business
ALT	The alias type key of the business (usually an acronym)
IDT	The industry type key of the business
AST	The association type key of the business
ORT	The organization type key of the business
NF	A generic term not recognized as a business name component

Business Name Standardization and Sun Master Index

Master index applications rely on the Master Index Standardization Engine to process business data. To ensure correct processing of business information, you need to customize the Matching Service for the master index application according to the rules defines for the standardization engine. This includes modifying mefa.xml to define parsing and phonetic encoding of the appropriate fields. You can modify mefa.xml using the Master Index Configuration Editor.

Standardization is defined in the StandardizationConfig section of mefa.xml, which is described in detail in "Match Field Configuration" in *Sun Master Index Configuration Reference* . To configure the required fields for parsing and normalization, modify the standardization structure in mefa.xml. To configure phonetic encoding, modify the phonetic encoding structure.

Generally, the BusinessName data type processes data that requires parsing prior to processing. You should not need to configure fields to normalize for business names. The following topics provide information about the fields used in processing business names and how to configure standardization for a master index application. The information provided in these topics is based on the default configuration.

- "Business Name Processing Fields" on page 75
- "Configuring a Standardization Structure for Business Names" on page 76
- "Configuring Phonetic Encoding for Business Names" on page 78

Business Name Processing Fields

When standardizing free-form business names, not all fields in a record need to be processed by the Master Index Standardization Engine. The standardization engine only needs to process fields that must be parsed, normalized, or phonetically converted. For a master index application, these fields are defined in mefa.xml, and processing logic for each field is defined in the Standardization Engine node configuration files.

Business Name Standardized Fields

The Master Index Standardization Engine expects that business name data will be provided in a free-form text field containing several components that must be parsed. By default, the match engine is configured to parse these components, and to normalize and phonetically encode the business name. You can specify additional fields for phonetic encoding.

If you specify the BusinessName match type for any field in the wizard, a standardization structure for that field is defined in mefa.xml. The fields listed under"Business Name Object Structure" on page 75 are automatically defined as the target fields. If you do not specify business name fields for matching in the wizard but want to standardize the fields, you can create a standardization structure in mefa.xml

Business Name Object Structure

For the default configuration of the BusinessName data type, the name field specified for standardization is parsed into several additional fields, one of which is also normalized. If you specify the BusinessName match type in the wizard, the following fields are automatically added to the object structure and database creation script.

- *field_name*_Name

- *field_name*_NamePhon
- *field_name*_OrgType
- *field_name*_AssocType
- *field_name*_Industry
- *field_name*_Sector
- *field_name*_Alias
- *field_name*_Url

  where *field_name* is the name of the field for which you specified business name matching. For example, if you specify the BusinessName match type for the Company field, the fields automatically added to the structure include Company_Name, Company_NamePhon, Company_OrgType, and so on.

You can add these fields manually if you do not specify a match type in the wizard.

Configuring a Standardization Structure for Business Names

For free–form business name fields, the source fields you define for parsing should include the standardization components that are predefined for parsing and normalization. For example, fields containing business information can include any of the field components listed in “Business Name Standardization Components” on page 61. The target fields can include any of these parsed fields. Follow the instructions under “Defining Master Index Standardization Rules” in *Sun Master Index Configuration Guide* to define fields for standardization. For the *standardization-type* element, enter **`BusinessName`**. For a list of field IDs to use in the *standardized-object-field-id* element, see “Business Name Standardization Components” on page 61.

Note – In the default configuration, the rules defined for the address data type assume that all input fields must be parsed as well as normalized. Thus, there is no need to configure fields only for normalization.

A sample standardization structure for business names is shown below. This structure parses a business name field into these standard business name fields: name, organization type, association type, sector, industry, and URL. Note that there is no domain selector specified, which would normally default to the United States domain; however, since business names are not variant dependent, it is irrelevant here.

```
<free-form-texts-to-standardize>
   <group standardization-type="BusinessName">
      <unstandardized-source-fields>
```

```
         <unstandardized-source-field-name>Company.Name
         </unstandardized-source-field-name>
      </unstandardized-source-fields>
      <standardization-targets>
         <target-mapping>
            <standardized-object-field-id>PrimaryName
            </standardized-object-field-id>
            <standardized-target-field-name>Company.Name_Name
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>OrgTypekeyword
            </standardized-object-field-id>
            <standardized-target-field-name>Company.Name_OrgType
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>AssocTypeKeyword
            </standardized-object-field-id>
            <standardized-target-field-name>Company.Name_AssocType
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>IndustrySectorList
            </standardized-object-field-id>
            <standardized-target-field-name>Company.Name_Sector
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>IndustryTypeKeyword
            </standardized-object-field-id>
            <standardized-target-field-name>Company.Name_Industry
            </standardized-target-field-name>
         </target-mapping>
         <target-mapping>
            <standardized-object-field-id>Url
            </standardized-object-field-id>
            <standardized-target-field-name>Company.Name_URL
            </standardized-target-field-name>
         </target-mapping>
      </standardization-targets>
   </group>
</free-form-texts-to-standardize>
```

Configuring Phonetic Encoding for Business Names

When you match or standardize on business name fields, the business name field should be specified for phonetic conversion (by default, the wizard defines this for you). Follow the instructions under "Defining Phonetic Encoding for the Master Index" in *Sun Master Index Configuration Guide* to define fields for phonetic encoding.

A sample of the *phoneticize-fields* element is shown below. This sample only converts the business name. You can define additional fields for phonetic encoding.

```
<phoneticize-fields>
   <phoneticize-field>
      <unphoneticized-source-field-name>Company.Name_Name
      </unphoneticized-source-field-name>
      <phoneticized-target-field-name>Company.Name_NamePhon
      </phoneticized-target-field-name>
      <encoding-type>NYSIIS</encoding-type>
   </phoneticize-field>
</phoneticize-fields>
```

Custom FSM–Based Data Types and Variants

The finite state machine framework of the Master Index Standardization Engine is very flexible, allowing you to define new data types and variants so you can standardize any type of data. This process requires no Java coding; all processing rules and logic are defined in XML files using predefined rules. The new data types and variants can be imported into NetBeans for use in master index projects. The following topics provide information and instructions for creating custom data types and variants.

- "About Custom FSM–Based Data Types and Variants" on page 78
- "About the Standardization Packages" on page 79
- "Creating Custom FSM-Based Data Types" on page 79
- "Creating Custom FSM-Based Variants" on page 83

About Custom FSM–Based Data Types and Variants

Creating a custom FSM data type or variant for the Master Index Standardization Engine requires defining the processing logic for the data type in an XML file. No Java coding is required in order to incorporate the comparators into a master index application. The processing logic is based in the files described in "Finite State Machine Framework Configuration" on page 15.

You define the following information for each data type or variant you create.

- The state model that defines each state, its input and output symbols, and transitions
- Any preprocessing, matching, or postprocessing logic for input and output symbols
- Any cleansing rules to be applied to the data prior to parsing
- Optionally, lists of non-standard values and the standard values to which they should be converted (such as a nickname table)
- Optionally, lists of possible values for a field component that helps the standardization engine identify and parse the component

After you create the package, you can import the custom data type or variant into NetBeans using the easy import function of Sun Master Index. You can then define standardization and normalization structures for the master index using the new data type or variant.

About the Standardization Packages

After you create a custom data type or variant you need to package the files in a ZIP file so they are available for import into NetBeans. Create a single package for each data type or variant.

For a custom data type, the ZIP file includes the following:

- A service type definition file
- One or more service instance definition files (depending on how many variants you include)
- One or more process definition file (`standardizer.xml`)
- Normalization files (optional)
- Lexicon files (optional)

For a custom variant, the ZIP file includes the following:

- One service instance definition file
- One process definition file (`standardizer.xml`)
- Normalization files (optional)
- Lexicon files (optional)

Creating Custom FSM-Based Data Types

You can define new data types and their corresponding variants using the flexible FSM framework of the standardization engine. Data types are easily incorporated into a master index project and can be made globally available to all projects. Perform the following steps to define a custom data type for the standardization engine.

- “Creating the Working Directory” on page 80

- “Defining the Service Type” on page 81
- “Defining the Variants” on page 81
- “Packaging and Importing the Data Type” on page 82
- “Service Type Definition File” on page 83

Creating the Working Directory

The working directory for custom data types requires a specific structure. At a minimum, the working directory will look similar to the following:

```
/WorkingDir
   serviceType.xml
   /lib
   /instance
      /Generic
         serviceInstance.xml
         /resource
            standardizer.xml
```

If the date type has several variants, the directory structure will not include the `Generic` folder, but will contain several folders named by the variants name in its place. Each variant folder must be of the same structure as the `Generic` folder shown above. The `resource` directory might also contain several normalization and lexicon files.

▼ To Create the Working Directory

1 **Create a working directory and add a `lib` and an `instance` directory at the top level.**

2 **Copy the files `standardizer-api.jar` and `standardizer-impl.jar` from `/`*NetBeans_Home*`/soa2/modules/ext/mdm/standardizer/lib` to the `lib` directory.**

3 **Do one of the following:**

- **If the data type only has one variant, create the following directory structure in the `instance` directory:**

  `/Generic/resource/`

- **If the data type has several variants, create the following directory structure in the `instance` directory for each variant:**

  `/`*VariantName*`/resource/`

4 **Continue to “Defining the Service Type” on page 81.**

Defining the Service Type

The `serviceType.xml` file defines information about the data type, and is a required file for each data type.

▼ To Define the Service Type

1 Create a file named `serviceType.xml` in your working directory.

Tip – You can copy the service type file from an existing data type and modify it for your use.

2 Enter text similar to the following, where *description* is the name of the data type and the *value* elements list the tokens, or standardization components, of the data type.

```
<serviceType configurationResource="standardizer.xml">
  <description>My Data Type Standardization</description>
  <parameter name="fields">
    <list>
      <value>Data Field1</value>
      <value>Data Field2</value>
      ...
    </list>
  </parameter>
</serviceType>
```

Note – For more information about the elements in this file, see "Service Type Definition File" on page 83.

3 Save and close the file.

4 Continue to "Defining the Variants" on page 81.

Defining the Variants

For each data type you create, you need to create one or more variants that define the logic for processing a specific type of data.

▼ To Define the Variants

Perform the following steps for each variant that will be used for the data type you are creating.

1 Define the service instance, as described in "Defining the Service Instance" on page 84.

Create the `serviceInstance.xml` file in `/`*WorkingDir*`/instance/`*VariantName*.

2 **Define the state model and processing logic, as described in "Defining the State Model and Processing Rules" on page 85.**

Create the `standardizer.xml` file in `/`*WorkingDir*`/instance/`*VariantName*`/resource`.

3 **If needed, create normalization and lexicon files, as described in "Creating Normalization and Lexicon Files" on page 87.**

Create the files in `/`*WorkingDir*`/instance/`*VariantName*`/resource`.

4 **Continue to "Packaging and Importing the Data Type" on page 82.**

Packaging and Importing the Data Type

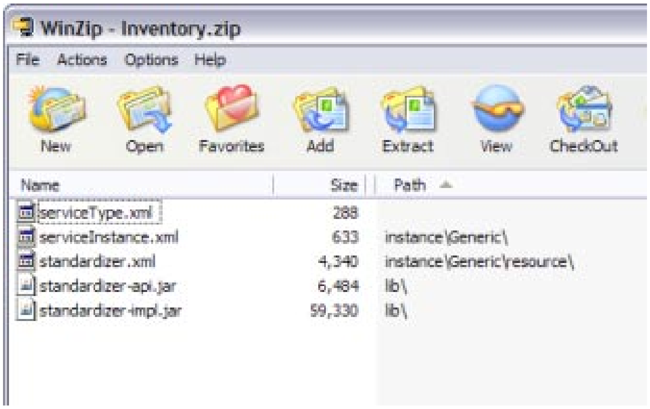
Once you have created all the files for the data type, you need to package them into a ZIP file to be imported into a master index application.

▼ To Package and Import the Data Type

1 **In the working directory, select the folders and files at the top level and add them to a ZIP file.**

2 **Name the ZIP file the same name as the data type.**

The ZIP file structure should like similar to the following:



Name	Size	Path
serviceType.xml	288	
serviceInstance.xml	633	instance\Generic\
standardizer.xml	4,340	instance\Generic\resource\
standardizer-api.jar	6,484	lib\
standardizer-impl.jar	59,330	lib\

FIGURE 1 Custom Data Type Zip File

3 **Import the file into a master index application as described in "Importing Standardization Data Types and Variants" in *Sun Master Index Configuration Guide*.**

Service Type Definition File

Each data type is configured by a service type definition file, `serviceType.xml`. Service type files define the fields to be standardized for a data type. The following table lists and describes the elements in the service type file.

Element	Attribute	Description
serviceType		A description and any parameters for the data type.
	configurationResource	The name of the standardization process file that defines the states and processing for the data type.
description		A brief description of the data type, such as "Address Standardization".
parameter		A parameter for the configuration resource. By default, the name of the parameter is "fields", and it is populated with a list of standardized field component names.
	name	The name of the parameter.
	value	One or more values for the parameter.

Creating Custom FSM-Based Variants

The flexible framework of the Master Index Standardization Engine allows you to define new FSM-based variants on existing FSM-based data types so you can standardize different categories of the same type of data. For example, you might need to standardize names from several different countries. Variants are easily incorporated into a master index project and can be made globally available to all projects. Perform the following steps to create a custom variant.

- "Creating the Working Directory" on page 83
- "Defining the Service Instance" on page 84
- "Defining the State Model and Processing Rules" on page 85
- "Creating Normalization and Lexicon Files" on page 87
- "Packaging and Importing the Variant" on page 88
- "Service Instance Definition File" on page 89

Creating the Working Directory

The working directory for custom variants requires a specific structure. At a minimum, the working directory will look similar to the following:

```
/WorkingDir
   serviceInstance.xml
```

```
  /resource
     standardizer.xml
```

The resource directory might also contain several normalization and lexicon files.

▼ To Create the Working Directory

1 **Create a working directory for the new variant.**

2 **In the new working directory, create a `resource` directory.**

3 **Continue to "Defining the Service Instance" on page 84.**

Defining the Service Instance

The `serviceInstance.xml` file for each variant defines the name of the variant, the data type it modifies, and additional Java class information.

▼ To Define the Service Instance

1 **Create a file named `serviceInstance.xml` at the top level of your working directory.**

Tip – You can copy a service instance file from an existing variant in the data type to which you will add the new variant, and then modify it for the new variant.

2 **Define values for the elements and attributes described in "Service Instance Definition File" on page 89.**

This example defines a new Spanish variant to the PersonName data type.

```
<serviceInstance type="PersonName">
  <description>Person Name Standardization: Spain</description>
  <parameter name="dataType" value="PersonName" />
  <parameter name="variantType" value="SP" />
  <componentManagerFactory
       class="com.sun.inti.components.component.BeanComponentManagerFactory">
    <property name="stylesheetURL"
       value="classpath:/com/sun/mdm/standardizer/impl/standardizer.xsl"/>
    <property name="urlSource" >
      <bean class="com.sun.inti.components.url.ResourceURLSource">
        <property name="resourceName" value="standardizer.xml />
      </bean>
    </property>
  </componentManagerFactory>
</serviceInstance>
```

Note – The value you enter for the *variantType* parameter must match the name you want the variant to display in the Standardization folder of the master index project.

3 **Save and close the file.**

4 **Continue to "Defining the State Model and Processing Rules" on page 85.**

Defining the State Model and Processing Rules

The state model defines how the data is read, tokenized, parsed, and modified during standardization. The state model and processing rules are all defined in the `standardizer.xml` file.

Before you begin this step, determine the different forms in which the data to be standardized can be presented and how it should be standardized for each form. For example, name data might be in the form "First Name, Last Name, Middle Initial" or in the form "First Name, Middle Name, Last Name". You need to account for each possibility. Determine each state in the process, and the input and output symbols used by each state. It might be useful to create a finite state machine model, as shown below. The model shows each state, the transitions to and from each state, and the output symbol for each state.

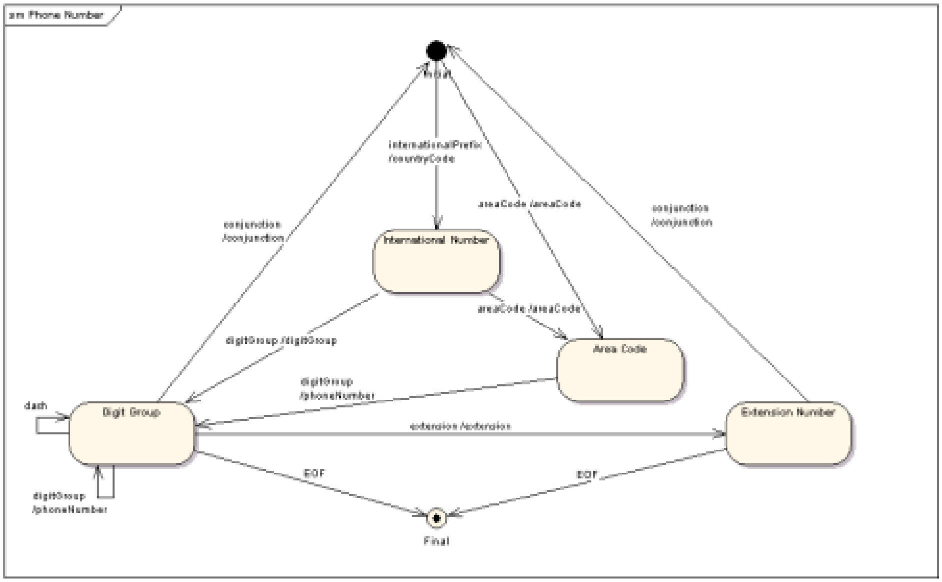


FIGURE 2 Sample Finite State Machine Model

For more information about the FSM model, see "FSM Framework Configuration Overview" on page 16.

To Define the State Model and Processing Rules

1 **In /*WorkingDirectory*/`resource`, create a new XML file named `standardizer.xml`.**

Tip – You can copy the file from an existing variant in the data type to which you are adding the custom variant. Then you can modify the file for the new variant.

2 **If the data you are processing does not need to be parsed, but only needs to be normalized, define normalization rules in the `normalizer` section of the file.**

For more information, see "Data Normalization Definitions" on page 23 and "Standardization Processing Rules Reference" on page 24.

3 **If the data you are processing needs to be parsed and normalized, define the state model in the upper portion of the file.**

For information about the state model and the elements that define it, see "Standardization State Definitions" on page 17.

Note – The next several steps use the processing rules described in “Standardization Processing Rules Reference” on page 24. Some of these rules might require that you create normalization and lexicon files.

4 **In the `inputSymbols` section of the file, define each input symbol along with any processing rules.**

For more information, see “Input Symbol Definitions” on page 19.

5 **In the `outputSymbols` section of the file, define each output symbol along with any processing rules.**

For more information, see “Output Symbol Definitions” on page 21.

6 **In the `cleanser` section of the file, define any cleansing rules that should be performed against the data prior to tokenization.**

For more information, see “Data Cleansing Definitions” on page 22.

7 **If you created any rules that reference normalization or lexicon files, continue to “Creating Normalization and Lexicon Files” on page 87.**

Creating Normalization and Lexicon Files

Lexicon files list the possible values for a field so the standardization engine can quickly and accurately recognize different field components. Normalization files list the nonstandard values that might be found in a field along with the standard version so the standardization engine can present a common form for the data. You need to create a file for each lexicon or normalization file you referenced from `standardizer.xml`.

For more information about normalization and lexicon files, see “Lexicon Files” on page 29 and “Normalization Files” on page 30.

▼ To Create Normalization and Lexicon Files

1 **For each normalization file you referenced in `standardizer.xml`, do the following:**

a. **Create a text file in `/`*WorkingDirectory*`/resource`.**

b. **Save the file under the name you used to reference it from `standardizer.xml`.**

c. **In the file, enter a list of nonstandard values along with their standardized values, separating the nonstandard value from the standard value with a pipe (|) as shown below.**

```
COR|COURT
CRT|COURT
CR.|COURT
CT|COURT
CT.|COURT
DR|DRIVE
DR.|DRIVE
DRV|DRIVE
...
```

d. **When you are finished, save and close the file.**

2 **For each lexicon file you referenced in `standardizer.xml`, do the following:**

a. **Create a text file in** `/`*WorkingDirectory*`/resource`**.**

b. **Save the file under the name you used to reference it from `standardizer.xml`.**

c. **In the file, enter a list of all possible values for the field as shown below.**

```
E
EAST
ET
N
NO
NORTH
NTH
S
SO
SOUTH
...
```

d. **When you are finished, save and close the file.**

3 **Continue to "Packaging and Importing the Variant" on page 88.**

Packaging and Importing the Variant

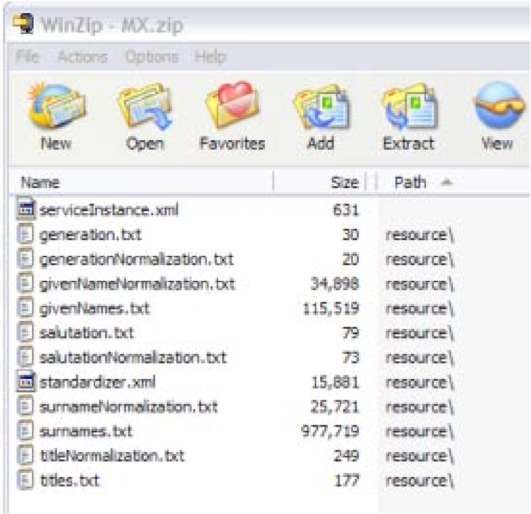
Once you have created all the files for the variant, you need to package them into a ZIP file to be imported into a master index application.

▼ To Package and Import the Variant

1. **In the working directory, select the folder and file at the top level and add them to a ZIP file.**

2. **Name the ZIP file the same name as the variant. This is the value you entered for the** *variantType* **parameter in "Defining the Service Instance" on page 84.**

   The ZIP file structure should be similar to the following. Note that this variant includes several normalization and lexicon files. Your variant might not contain any.



Name	Size	Path
serviceInstance.xml	631	
generation.txt	30	resource\
generationNormalization.txt	20	resource\
givenNameNormalization.txt	34,898	resource\
givenNames.txt	115,519	resource\
salutation.txt	79	resource\
salutationNormalization.txt	73	resource\
standardizer.xml	15,881	resource\
surnameNormalization.txt	25,721	resource\
surnames.txt	977,719	resource\
titleNormalization.txt	249	resource\
titles.txt	177	resource\

FIGURE 3 Custom Variant Zip File

3. **Import the file into a master index application as described in "Importing Standardization Data Types and Variants" in *Sun Master Index Configuration Guide*.**

Service Instance Definition File

Each data type variant is configured by a service definition file. Service type files define the fields to be standardized for a data type, and service instance definition files define the variant and Java factory class for the variant. Both files are in XML format.

Element	Attribute	Description
serviceInstance		A container element for the description and any parameters for the variant.

Element	Attribute	Description
	type	The name of the data type to which the variant belongs.
description		A brief description of the variant, such as "Person Names: Spain".
parameter		One parameter for the variant. The default variants contain two parameters, *dataType* and *variantType*. The *dataType* parameter specifies the name of the data type to which the variant belongs. The *variantType* parameter specifies the name of the variant. For a master index application, these are the names of the nodes that appear under the Standardization Engine node.
	name	The name of the parameter.
	value	The value of the parameter.
componentManagerFactory		The component manager factory class for the variant.
	class	The name of the component manager factory class. The default class is `com.sun.inti.components.component.BeanComponentManagerFacto`
property		A property of the component manager factory class. The default class has two properties. The `stylesheetURL` property defines the location of the stylesheet, `standardizer.xml`. The `urlSource` property defines the process definition file. Its value is a bean (by default, `com.sun.inti.components.url.ResourceURLSource`), which has a property called `resourceName`. The value for this property is `standardizer.xml`.
	name	The name of the property.
	value	The value for the property.