

ArcGIS® 9

Geocoding Rule Base Developer Guide



Copyright © 2003 ESRI
All rights reserved.
Printed in the United States of America.

The information contained in this document is the exclusive property of ESRI. This work is protected under United States copyright law and other international copyright treaties and conventions. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system, except as expressly permitted in writing by ESRI. All requests should be sent to Attention: Contracts Manager, ESRI, 380 New York Street, Redlands, CA 92373-8100, USA.

The information contained in this document is subject to change without notice.

CONTRIBUTING WRITERS
Agatha Tang and Kristin Clark

U.S. GOVERNMENT RESTRICTED/LIMITED RIGHTS

Any software, documentation, and/or data delivered hereunder is subject to the terms of the License Agreement. In no event shall the U.S. Government acquire greater than RESTRICTED/LIMITED RIGHTS. At a minimum, use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR §52.227-14 Alternates I, II, and III (JUN 1987); FAR §52.227-19 (JUN 1987) and/or FAR §12.211/12.212 (Commercial Technical Data/Computer Software); and DFARS §252.227-7015 (NOV 1995) (Technical Data) and/or DFARS §227.7202 (Computer Software), as applicable. Contractor/Manufacturer is Environmental Systems Research Institute, Inc., 380 New York Street, Redlands, CA 92373-8100, USA.

ESRI, ArcIMS, MapObjects, ArcView, Avenue, StreetMap, ArcCatalog, ArcMap, ArcSDE, ArcLogistics, SDE, Spatial Database Engine, and ArcToolbox are trademarks, registered trademarks, or service marks of ESRI, registered in the United States, the European Community, or certain other jurisdictions.

The names of other companies and products herein are trademarks or registered trademarks of their respective trademark owners.

Contents

1	Introduction	1
	What is geocoding?	2
	Components in the geocoding process	3
	General process of geocoding	4
	How to use this book	8
2	Overview of the rule base	9
	The standardization process	10
	MatchRules	12
3	The match file	13
	Overview of the match file	14
	VAR commands	15
	MATCH commands	16
	VARTYPE commands	18
	m and u probabilities	19
	Matching weights	20
	Modifying the .mat file	21
4	The command file	23
	Overview of the command file	24
	Parsing parameters	25
	Adjusting parsing parameters	26
	Adding the DEBUG and OUTFILE keywords to the us_addr.stn file	27
5	The match key dictionary	29
	Overview of the match key dictionary	30
	Modifying the match key dictionary	32

6 The classification table 35

- Overview of the classification table 36
- How to get to the classification table 37
- How the classification table is formatted 38
- Modifying the .cls file 40

7 The pattern file 45

- Overview of the pattern file 46
- Pattern rules 48
- Actions 55
- Modifying the pattern file 60
- Dealing with street intersections 64
- Editing intersection .xat/.pat files 66
- Adding custom routines to the pattern file 67

8 Developer's Kit tools 69

- STANEDIT and the DEBUG and OUTFILE keywords 70
- Creating a new process with STANEDIT 72
- ENCODEPAT 73
- What to do before adding your files to the folder 74

Appendix A: Data dictionaries and match rules syntax 75

- Introduction 76
- Data dictionaries and match specifications 77
- Geocoding 87
- Record linkage concepts 88

Appendix B: Standardization process syntax 93

- Introduction 94
- Input file format specifications 96
- The match key 103
- The classification table 107
- The pattern rules 110
- Unconditional patterns 114
- Conditional patterns 121
- Actions 129
- Summary 142

Appendix C: Developer's Kit software tools reference 143

- Developer's Kit software tools reference 144

Appendix D: Matching and standardization file conventions and limits 149

- Conventions and limits 150

Glossary 151

Index 163

Introduction

1

IN THIS CHAPTER

- **What is geocoding?**
- **Components in the geocoding process**
- **General process of geocoding**
- **How to use this book**

If you are reading this book, chances are you have learned quite a bit about using ESRI® geocoding products, such as ArcGIS®, ArcIMS®, MapObjects®, ArcView® 3.x, and so on, and are now ready to do some customization of the rule base to make the applications even more useful to you.

This book will walk you through concepts about the Geocoding Developer's Kit and tasks that will show you how to take full advantage of the kit. Each chapter, with the exception of Chapter 2, which provides an overview of the rule base, focuses on a specific aspect of the kit. For example, Chapter 3 looks at the match file, Chapters 4–7 focus on the files that make up the standardization process, and Chapter 8 examines the tools that come with the Geocoding Developer's Kit.

You can use this book as a reference for a specific problem or read through it to gain a complete understanding of the process of customizing the rule base. Either way, this book will be useful in helping you get the most out of your geocoding applications.

What is geocoding?

Geocoding, which is also known as address matching, is the process of assigning an x,y coordinate value to the description of a place by comparing the descriptive location elements to those present in the reference data. These x,y coordinates are points that can be displayed on a map.

Geocoding is useful because, in many cases, geographic data exists that describes locations such as street addresses, city names, ZIP Codes, or even telephone numbers. While humans understand what these descriptions mean and how they relate to locations on the earth's surface, computers do not. To display these locations on a map and perform analyses with them, a computer must be given geometric representations, such as point features, of these locations.

How the *Geocoding Rule Base Developer Guide* can help you

You can use the *Geocoding Rule Base Developer Guide* to learn how to modify the address standardization process and match rules. Address standardization is implemented using a pattern recognition method associated with lookup tables. Some tasks, such as adding a new street type value or modifying the abbreviation of a standardized street type, can be done by simply editing a text file. If you need to add or alter the standardization process of certain unusual addresses, you can learn and use the pattern recognition syntax described in Chapter 7, 'The pattern file', and start writing or modifying the patterns in the existing rule base file.

Components in the geocoding process

The geocoding process requires reference data, address data, and software. This section will discuss each of these components.

Reference data

Reference data is referred to as geographic base files (GBFs). It can vary from simple digitized boundary files to more complex address coding guides (ACGs) to even more sophisticated street centerline files.

Reference data is available from both public and private data publishers. The public data publishers include the U.S. Census Bureau TIGER and United States Postal Service (USPS) city/state, five-digit ZIP Codes, and ZIP+4 files. The private data publishers include Geographic Data Technology, Thomas Bros., NavTech, and Tele Atlas. In addition to the public and private data publishers, there are also some specialized GBFs. They include floodplains, telephone switch centers, and third-party address coding guides.

Address data

Address data types vary by application. Some examples are customer addresses, location of the event or incident (such as accidents, fires, crimes, and so on), location of equipment and facilities (such as pay phones, electrical transformers, convenience stores, and so on), and monument offset (for example, two miles from the intersection of State Route 57 and Main Street).

Typically, input records are captured without regard to standards or format. In addition, in most instances, records contain errors and omissions. These errors, however, may be corrected through a standardization process or address lookup from a reference database.

Software

Some systems have been designed and implemented for a specific geocoding discipline. Typically, these systems are integrated with a single GBF. These systems are hard-coded and difficult to fine-tune.

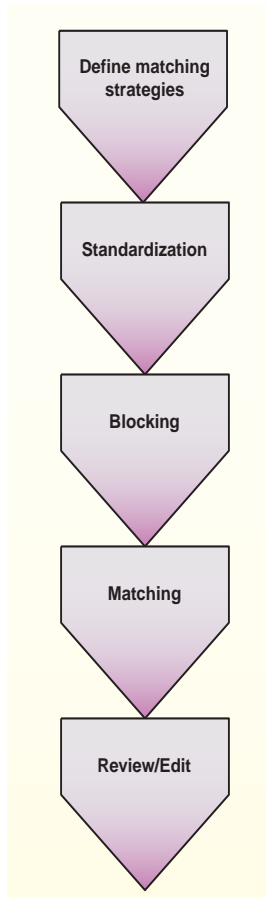
Probabilistic record linkage systems use a statistically valid form of fuzzy logic to score how well records do or do not match. This type of system allows for reviews of “almost” matches: fine-tuning of the matching rules, table, and cut-off thresholds.

Geocoding software needs to operate in both sequential batch mode and single event mode. It also needs to be able to integrate with other applications and operate transparently (because geocoding may be part of an interactive client/server environment) and be able to integrate easily with spatial analysis software.

ESRI’s geocoding applications are built on top of MatchWare Technology’s Probabilistic Record Linkage system. The applications incorporate fuzzy matching logic, are flexible and customizable, operate in both batch and single event modes, integrate well with other applications, and are integrated with a geographic information system (GIS).

General process of geocoding

The general process of geocoding is outlined in the chart below:



As the chart shows, the geocoding process has five stages. This section gives a brief introduction to the process.

Defining matching strategies

Matching strategies define the method of geocoding. When you are looking at matching strategies, there are certain questions you should ask yourself:

- What fields need to be indexed?
- What fields will be matched?
- What would you consider a match?
- How do you handle errors?
- How will you do manual review?
- What is the default functionality?

After you have answered these questions, you are ready to begin defining your matching strategies. When you define your matching strategies, you need to identify the fields that the style will match against. The reference data needs to contain these fields.

After you have identified these fields, you need to identify the fields used for *blocking*, which is a way to group information together for faster searching. Indexes will be built for the blocking fields. You also need to identify the address components for matching. Each applicable address component is called a *match key*. At a minimum, the match key should contain house number, prefix type, prefix direction, street name, street type, and zone fields. For more information about match keys, see Chapter 5, ‘The match key dictionary’.

Standardization

Standardization has two steps. The first step, preparing the reference file, includes separating the data into individual fields

and using standardized names or abbreviations. To secure a good match rate, all names or abbreviations should be considered across the database.

The second step, preparing the addresses for matching, includes breaking down the address data into individual fields and converting the values, if needed, with standard abbreviations such as “AVE” for “AVENUE”.

The following are some standardization examples:

380 New York St

380 = House number

New York = Street name

St = Street type

123 1 St St St 123

123 = House number

1 St = Street name

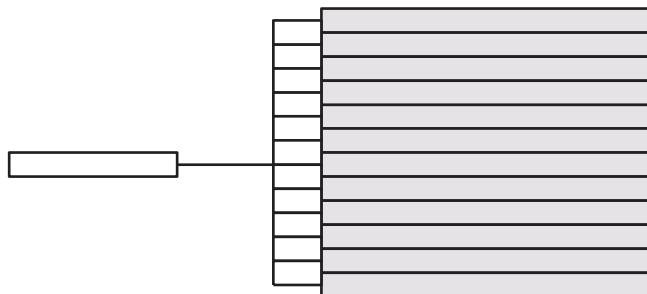
St = Street type

St 123 = Suite number

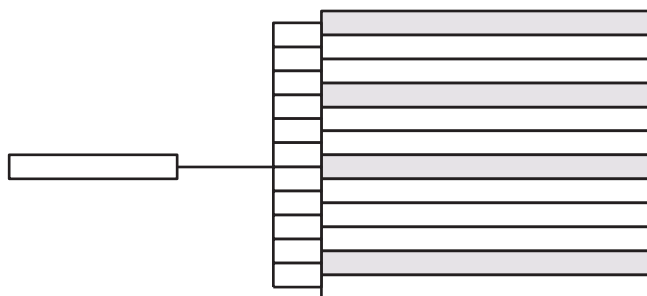
Blocking

Blocking, a way to group relevant information together, reduces the number of potential matches that need to be checked. It requires fast index lookup and an efficient index search method, such as Soundex.

Without blocking, every record is reviewed:



With blocking, only blocked records are reviewed:



Soundex

One common method of indexing names is to use Soundex. Soundex finds the match plus all the potential candidates. For example, you may want to search for Main Street. Soundex will find Main, as well as Memne. Soundex is particularly useful when you are working with names that are easily misspelled. Soundex encoding transforms a string into code that tends to bring

together all variants of the same string. In the example of Main and Memne above, both words would be coded M500.

For address matching, street name is often an important blocking variable. To maximize the chance that similarly spelled street names reside in the same index block, the Soundex system can be used to code the street names, and the Soundex code can be used as a blocking variable.

There are six steps to encoding a string using the Soundex method.

1. Retain the first letter of the string, and drop all occurrences of a, e, h, i, o, u, w, and y in other positions.
2. Assign the following numbers to the remaining letters after the first:
b, f, p, v: 1
c, g, j, k, q, s, x, z: 2
d, t: 3
l: 4
m, n: 5
r: 6
3. Ignore the spaces if the string contains multiple words.
4. If two or more letters with the same code were adjacent in the original string (before step 1), omit all but the first.
5. Convert to the form “letter, digit, digit, digit” by adding trailing zeros if there are less than three digits or by dropping the rightmost digits if there are more than three.
6. For any character that can’t be mapped, ESRI’s geocoding products either retain the character or skip to the next character for generating the Soundex code.

Matching

The software uses field-by-field comparisons to generate a detailed candidate score and to determine a good match. In field-by-field comparisons, the software compares each field of the candidate address to each field in the original address. For example, it compares the prefix direction in the address being searched to the prefix direction in the reference data.

There are several types of comparisons, including uncertainty character comparison and left/right interval comparisons. Uncertainty character comparison is usually used to compare street names and zones in order to allow more flexibility in matching even if the names are misspelled, which is likely to happen in street names and zones.

Left/Right interval comparisons compare the house number against number ranges in a format of two intervals: one for the low and high numbers on the left side of the street, and one for the low and high numbers on the right side of the street.

With all these types of comparisons, the closer the candidate fields are to the standardized address fields, the higher the candidate score. The candidate score is calculated based on the matching and unmatching probabilities, also called the m and u probabilities. For more information about the m and u probabilities, see Chapter 3, ‘The match file’.

A composite score is computed by summing the score contributed by each element. The score is normalized to a value between 0 and 100. The software ranks the candidates by the score and thus determines which candidate is a good match.

For more information on matching, see Chapter 3, ‘The match file’.

Review/Edit

You can fine-tune the geocoding process during the review/edit stage. In other words, you can adjust index search rules, adjust matching weights, and adjust minimum matching scores. By manipulating each of these parameters, you can get results that are the most useful to you. Once you have everything set up as you like, you will be able to find the candidates you want.

How to use this book

This book is intended for the geocoding user who wants to modify and customize the geocoding rule base. It is primarily intended to give conceptual and hands-on information about the Geocoding Developer's Kit. If you would like to review some of the basics of geocoding in ArcGIS, see *Geocoding in ArcGIS*.

As most of the ESRI GIS products, including ArcGIS, ArcSDE®, ArcIMS, MapObjects, ArcLogistics™ Route, and ArcView, use the same geocoding engine, modifying the rule base can be done once and applied to all these products. This book is not designed to go into depth about ArcGIS or MapObjects software; instead, it is a more generic and conceptual book about the Geocoding Developer's Kit. For more information about geocoding in specific applications, review the introductory documentation applicable to your specific application.

The overall goal of this book is to give you an understanding of how to use the Geocoding Developer's Kit and modify the rule base. It will introduce you to the tools included in the kit and show you specific tasks in modifying your geocoding rule base files to make them more useful to you. The concepts presented here and the tasks that accompany them are intended to give you a better understanding of how to manipulate the rule base—after you familiarize yourself with the concepts and tasks, you should be able to use the Geocoding Developer's Kit to perform advanced customization of your rule base.

Overview of the rule base

2

IN THIS CHAPTER

- The standardization process
- MatchRules

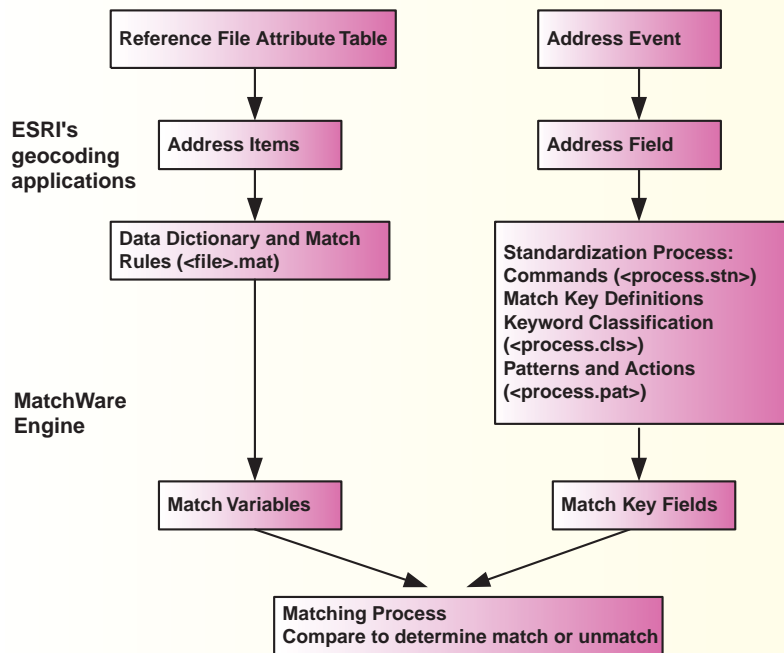
The *rule base* is a collection of files that direct the geocoding engine in how to standardize address data and match that data to the related location in the reference data. This chapter provides an overview of the rule base and what it comprises. The first section focuses on the *standardization process*, and the second section looks at the *match rules*. Later chapters go into great detail about the topics that are introduced here.

The standardization process

Standardization is a process of address parsing that prepares the address to be matched against the reference data. It breaks down the address into elements, assigns them to the *match keys*, and converts the abbreviated address elements into standardized values (for example, AVE for Avenue). Instead of looking in the reference data for one single string (the address), the standardization is essential for preparing the address data that allows the geocoding engine to search for possible candidates and matching the address more efficiently by using street name search and field-to-field comparison.

A standardization process contains standardization commands (<file>.stn), a *match key dictionary* for the event address (<file>.dct), a *classification table* for providing standard address abbreviations (<file>.cls), and patterns and *actions* for standardizing an address (<file>.pat). The standardization process can also contain additional tables, such as <file>.tbl, but they are optional. By making modifications in each of these files, you can alter the way the geocoding process is done.

Relationship between MatchRules and StanRules (standardization process)



A standardization process *tokenizes* and *classifies* the components of an address. In the standardization process, standard abbreviations are substituted for keywords. Identified address components are then moved into different match key fields.

How your *reference file* data is set up will have an impact on your results. Geocoding comparisons are not case-sensitive. However, the data in your reference file should be prestandardized. In addition, each address element should be stored in a separate field. For information on the required format of your reference data, see *Geocoding in ArcGIS*.

The standardization commands (.stn extension) specify input record size, debug mode, output file name, and the process name. For more information about the *command file*, see Chapter 4, ‘The command file’.

The match key dictionary (.dct extension) defines the data type, field length, and missing value code for each match key field. For a detailed look at the *match key dictionary*, see Chapter 5, ‘The match key dictionary’.

The classification table (.cls extension) interprets various keywords found in an address and provides a standard abbreviation for each keyword. For a detailed look at the *classification table*, see Chapter 6, ‘The classification table’.

The pattern file (.pat extension) defines *pattern rules* and actions for standardizing the input record into the match key fields. For a detailed look at the *pattern file*, see Chapter 7, ‘The pattern file’.

MatchRules

MatchRules define the address fields from the reference data used for matching. They also specify the methods of address-to-reference data comparisons and the weight on matching an address element. As saved in a text file with a .mat file extension, such as us_addr1.mat, MatchRules are defined in a few different command lines. First, the *VAR commands* specify the variable names, field position, field length, and missing value codes for the address fields associated with the reference data.

When you are defining match rules for match variables, a number of *MATCH commands* are available for specifying the comparison type, match key field, variable name, matching probabilities, and additional parameters. For more information on MATCH commands, see Chapter 3, ‘The match file’.

MatchRules also specify probabilities for a score comparison. These probabilities are known as the *m* and *u probabilities*. The *m* probability is the probability that the field agrees given the record pair is a match. For example, if the prefix direction field is North in the original address and North in the candidate address, the *m* probability will give the score a higher value. The higher the *m* probability is, the more weight that particular field has when the candidate scores are computed. On the other hand, if the prefix direction field is West in the original address and North in the candidate address, the score will be much lower.

The *u* probability is the probability that the field agrees at random. This number is often much lower than the *m* probability, which can range from 0.1 to 0.999. Both the *m* and *u* probabilities are defined in the .mat file.

For more information on *m* and *u* probabilities, see Chapter 3, ‘The match file’.

In addition to VAR and MATCH commands, there is a special command called *VARTYPE*. This command ensures that no frequency analysis is performed. For more information on VARTYPE, see Chapter 3, ‘The match file’.

3

The match file

IN THIS CHAPTER

- Overview of the match file
- VAR commands
- MATCH commands
- VARTYPE commands
- m and u probabilities
- Matching weights
- Modifying the .mat file

The *match file* (.mat extension) is where *MatchRules* are defined, *m* and *u probabilities* are specified, and the weight of each *address element* is set. This chapter provides a brief overview of the match file, then explains in detail what each part of the match file is and how to set it up. It also goes through some tasks to show you how you can modify the match file to adjust weights and m and u probabilities.

Overview of the match file

MatchRules define the variables for the address items found in the reference street file attribute table. MatchRules use *VAR* commands to specify variable names, field position, field length, and missing value codes. You need a .mat file extension to specify match rules—for example, us_addr1.mat.

The VAR variable names can't contain more than 16 characters, and the first character in the name must be a letter of the alphabet. In addition, names are not case-sensitive, spaces aren't allowed, but underscores are permitted.

The Match (.mat) file

```
; @ (#) us_addr1.mat
;
; Full geocoding match rules with left and right zip codes
;
VAR LeftFrom    1 10 X ; Left from house number
VAR LeftTo     11 10 X ; Left to house number
VAR RightFrom  21 10 X ; Right from house number
VAR RightTo   31 10 X ; Right to house number
VAR PreDir    41  2 X ; Prefix direction
VAR PreType   43  6 X ; Prefix street type
VAR StreetName 49 30 S ; Street name
VAR StreetType 79  6 X ; Suffix street type
VAR SufDir    85  2 X ; Suffix direction
VAR LeftZone  87 20 X ; Left zone
VAR RightZone 107 20 X ; Right zone
;
MATCH LR_UNCERT ZN LeftZone RightZone 0.9 0.01 700.0 EITHER
MATCH UNCERT SN StreetName 0.9 0.01 700.0
MATCH CHAR PD PreDir 0.8 0.1
MATCH CHAR PT PreType 0.7 0.1
MATCH CHAR ST StreetType 0.85 0.1
MATCH CHAR SD SufDir 0.85 0.1
MATCH D_INT HN LeftFrom LeftTo RightFrom RightTo 0.999 0.05 ZERO_VALID
;
VARTYPE LeftFrom NOFREQ
```

VAR commands (lines 10-20)

MATCH commands (lines 25-35)

VARTYPE command (line 40)

u probability (points to 0.7 in MATCH CHAR PT PreType 0.7 0.1)

m probability (points to NOFREQ in VARTYPE LeftFrom NOFREQ)

VAR commands

VAR commands specify variable names, field position, and missing value codes in the match file. As the graphic on the previous page shows, VAR commands are listed in the match file and are always prefaced with VAR.

Format

The format of the VAR command in the MatchRules should be as follows:

```
VAR  
<variable-name>  
<beginning-column>  
<length>  
<missing-value code>  
; comments
```

where <missing-value code> ::=

- S—spaces
- Z—zero or spaces
- N—negative number (for example, -1)
- 9—all nines (for example, 9999)
- X—no missing value

An example of this is:

```
VAR StreetName 37 28 S ; Street name
```

In this example, the variable StreetName begins in column 37 for length 28. The space or spaces represent valid missing values. The text that follows a semicolon (;) is considered to be a comment.

MATCH commands

You should use *MATCH commands* when you are defining match rules for match variables. A number of MATCH commands are available for specifying the comparison type, *match key* field, variable name, matching probabilities, and additional parameters.

Format

The format of the MATCH commands should be as follows:

```
MATCH
<comparison-type>
<match key field>
<reference file variable name>
<m-probability>
<u-probability>
[<additional parameters>]
[<mode>]
```

where <comparison-type> :=

CHAR—Character
D_INT—Left/Right intervals
LR_CHAR—Left/Right character string comparison
LR_UNCERT—Left/Right uncertainty string comparison
NUMERIC—Numeric
UNCERT—Uncertainty character comparisons
INTERVAL_NOPAR—Interval without parity

Match key field

The two-character match key field is defined in the *match key dictionary* (.dct) file.

Reference file variable name

The variable name is defined in the VAR command.

m probability

The m probability is the probability that the field agrees, given the record pair is a match (one minus the error rate of the field in a matched record).

u probability

The u probability is the probability that the field agrees at random.

For more information on the m and u probabilities, see the ‘m and u probabilities’ section in this chapter.

Additional parameters

A numeric value or values required for types UNCERT and LR_UNCERT:

900 indicates that the two strings are identical

850 indicates that the two strings are so close they can be safely considered the same

800 indicates that the two strings are probably the same

750 indicates that the two strings are probably different

700 indicates that the two strings are almost certainly different

Mode

The mode is required for types D_INT:

ZERO_VALID

ZERO_NULL

EITHER

ZERO_VALID indicates that zeros or blanks should be treated as any other number in that particular field.

For example, if ZERO_VALID appears in the {HN} field, as it does in the example on the next page, any zeros in the house number should be treated as an actual number.

ZERO_NULL indicates that zero is null and is never part of the interval.

EITHER is required for types LR_CHAR and LR_UNCERT. It indicates that it can match on either the left or right side. An example of EITHER is shown in the MATCH commands below.

```
MATCH LR_UNCERT ZN ZipLeft ZipRight 0.9 0.01  
700.0 EITHER
```

```
MATCH UNCERT SN StreetName 0.9 0.01 700.0
```

```
MATCH CHAR PD PreDir 0.8 0.1
```

```
MATCH CHAR PT PreType 0.7 0.1
```

```
MATCH CHAR ST SuffixDir 0.85 0.1
```

```
MATCH D_INT HN FromLeft ToLeft FromRight ToRight  
0.999 0.05 ZERO_VALID
```

VARTYPE commands

The *VARTYPE command* comes after the MATCH commands in the match file. The VARTYPE command indicates that frequency analysis isn't performed. When you are modifying or creating a match file, you shouldn't alter this line at all except to put in the appropriate variables.

Format

The format of the VARTYPE commands should be as follows:

VARTYPE

<match variable name>

<action>

where

<action> ::=

NOFREQ (indicates that no frequency analysis should be performed)

An example of this is:

```
MATCH LR_UNCERT ZN ZipLeft ZipRight 0.9 0.01
700.0 EITHER
MATCH UNCERT SN StreetName 0.9 0.01 700.0
MATCH CHAR PD PreDir 0.8 0.1
MATCH CHAR PT PreType 0.7 0.1
MATCH CHAR ST SuffixDir 0.85 0.1
MATCH D_INT HN FromLeft ToLeft FromRight ToRight
0.999 0.05 ZERO_VALID
```

```
VARTYPE FromLeft NOFREQ
```

In this example, FromLeft is the variable. The example also shows that the VARTYPE command comes after the MATCH commands—it should be the last command in the match file.

m and u probabilities

MATCH commands require m and u probabilities.

The m probability is the probability that the field agrees to the corresponding field in the standardized address, given the record pair is a match. It can be defined as 1 minus the error rate of the field in a matched record. Given this definition, if StreetName disagrees 10 percent of the time in a sample of matched records (for example, because of a transcription error or being misreported), then the m probability for this variable is 0.9 (1 – 0.1). The more reliable a field is, the greater the m probability will be.

You should provide an initial estimate of the m probability. Values of 0.9 to 0.99 are typical, although any value from 0.1 through 0.999 is allowed. The closer this value is to one, the more critical a disagreement on the field becomes. This means you can set fields that are important to have a high penalty for mismatching. In your initial estimates of the m probability, the estimates should tend to be high. The fields that are the most important should have the highest m probabilities associated with them, but you can have the same values for each field if you want to give each field equal weight. For more information on adjusting the m and u probabilities, see ‘Modifying the .mat file’ in this chapter.

The u probability is the probability that the field agrees to the corresponding field in the standardized address, given the record pair is unmatched. In other words, it is the probability that the field agrees at random. The probability that the State (U.S.) variable agrees at random is about 0.0004 if the State field in the dataset contains all 50 unique values (50 states in the U.S.). Given a uniform distribution, there are 2,500 (50 * 50) possibilities. The State agrees in 1 of the 2,500 combinations (thus, 0.0004 u probability).

In addition to providing an initial estimate of the m probability, you should also estimate a u probability. In general, your estimates here should be low, such as 0.01 or 0.1.

You can see an example of the m and u probabilities in the following code sample. The m probability is the first number immediately after each address element, and the u probability is the number following the m probability.

```
MATCH LR_UNCERT ZN ZipLeft ZipRight 0.9 0.01
700.0 EITHER
MATCH UNCERT SN StreetName 0.9 0.01 700.0
MATCH CHAR PD PreDir 0.8 0.1
MATCH CHAR PT PreType 0.7 0.1
MATCH CHAR ST SuffixDir 0.85 0.1
MATCH D_INT HN FromLeft ToLeft FromRight ToRight
0.999 0.05 ZERO_VALID
```

Matching weights

Once you have estimated the probabilities for each field, you can calculate the weight of each address element. If the fields agree, the weight is the log to the base 2 of the ratio of the m and u probabilities (positive weight). The equation is:

$$\log_2 m/u$$

If the fields disagree, then the weight is the log to the base 2 of the ratio of 1-m and 1-u (negative weight). The equation is:

$$\log_2 1-m/1-u$$

A composite weight is computed by adding the individual weights for all field comparisons together. The *composite weight* provides users with a reference for how good a match is and from

this, a candidate score is computed. ESRI's geocoding software products multiply the composite weight by 100 and use the result as the candidate score. Each set of matched rules may generate different scores depending on the number of rules defined and the specifications of the m and u probabilities. It is important to normalize the scores so that the perfectly matched candidate always shows a score of 100. You have to specify the maximum score as a factor for normalization in the address style file in ArcView 3.x. However, ESRI's other geocoding products, such as MapObjects, ArcIMS, and ArcGIS, normalize the score internally. The score of each matched candidate is *normalized* to the range of 0–100 based on the computed maximum score.

Examples of candidate scoring

Candidates	Composite Score
101 199 N MAIN ST + + + + +	100
101 199 MAIN ST + + - + +	90
101 199 N MAIN AVE + + + + -	85
101 199 MAIN + + - + -	60

This table shows examples of candidate scoring based on the event 123 N MAIN ST. The higher the score, the better the match.

Modifying the .mat file

The match (.mat) file is used to adjust the weights of each address element in determining a *match score*. You can adjust the values based on the confidence you have in each element of the address.

In some instances, you may want one of the address elements to have a different weight when determining the match score. You can modify the m and u probabilities in the .mat file to adjust this.

For example, perhaps you aren't confident of the StreetName element. If this is the case, you may want to adjust the .mat file so that the StreetName element has less weight.

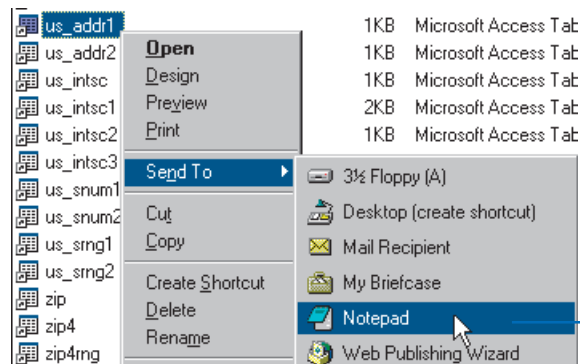
This task illustrates how to do this process using the us_addr.mat file. A similar process could be done using other .mat files.

This task has also been illustrated using a Windows interface. The basic text editing tasks could be performed in any text editing software package. It is advisable to make a copy of the original file, modify the copy, slightly change the name of the original file, and change the copy's name to match ►

Adjusting the weight of the StreetName element

1. Open the us_addr1.mat file in the geocode folder in a text editor, such as Notepad.

The .mat file will display. ►



```
MATCH LR_UNCERT ZN LeftZone RightZone 0.9 0.01
700.0 EITHER
MATCH UNCERT SN StreetName 0.9 0.01 700.0
MATCH CHAR PD PreDir 0.8 0.1
MATCH CHAR PT PreType 0.7 0.1
MATCH CHAR ST StreetType 0.85 0.1
MATCH CHAR SD SufDir 0.85 0.1
MATCH D_INT HN LeftFrom LeftTo RightFrom
RightTo 0.999 0.05 ZERO_VALID
```

The .mat file is displayed.

the original name. This way, if your modifications produce undesired results, you can restore the original file.

Tip

MatchRule syntax checker

The Developer's Kit doesn't contain a MatchRule syntax checker.

MatchRules must be visually verified for correctness before they are installed. If a MatchRule syntax error is detected when the MatchCase object is created, creation of the MatchCase object will fail.

2. Change the m probability of the StreetName to 0.8. This reduces how much the m probability adds to the score, thus reducing the weight of the StreetName element.
3. Click the File menu and click Save.

2

```
MATCH LR_UNCERT_ZN LeftZone RightZone 0.9 0.01
700.0 EITHER
MATCH UNCERT_SN StreetName 0.8 0.01 700.0
MATCH CHAR_PD PreDir 0.8 0.1
MATCH CHAR_PT PreType 0.7 0.1
MATCH CHAR_ST StreetType 0.85 0.1
MATCH CHAR_SD SufDir 0.85 0.1
MATCH D_INT_HN LeftFrom LeftTo RightFrom
RightTo 0.999 0.05 ZERO_VALID
```

The command file

4

IN THIS CHAPTER

- Overview of the command file
- Parsing parameters
- Adjusting parsing parameters
- Adding the DEBUG and OUTFILE keywords to the us_addr.stn file

The *command file* (.stn extension) is the file in the *standardization process* that defines the *standardization* commands and processes, such as us_addr.stn. This chapter gives an overview of the command file and shows you the format of the file and the commands within it. It also describes the *parsing parameters*, what they do, and how you can modify them.

Overview of the command file

The command file specifies the standardization commands and processes, such as `us_addr.stn`. The filename is always `<file>.stn`. The format of the command file should be:

RECORD <record-size>

The `<record-size>` is the size of the record in characters. ESRI geocoding products always use size 256.

TYPE <file-type>

The `<file-type>` is the type of file. Always use ASCII.

INTERACTIVE

Always use Interactive.

{DEBUG}

`{DEBUG}`, which is optional, is used to test the operation of the pattern matching. Information about which patterns were matched and what data was moved to the *match key* is printed either on the standard output or on an output file if specified. This statement has to be removed before the file is installed.

STANDARDIZE <process>

The `<process>` is the process name, for example, `us_addr`. The process name has to be the same name given to the `.stn`, `.dct`, `.cls`, and `.pat` files in the same process.

{OUTFILE <output-file>}

The `<output-file>`, which is optional, contains the results of the standardization, along with information about what patterns were

matched by each input case. This statement has to be removed before the file is installed in ESRI geocoding products.

{<Parsing parameters> }

The `{<Parsing parameters>}`, which are optional, are used to override the default. For more information, see the section ‘Parsing parameters’ in this chapter.

An example of this is:

```
us_addr.stn
RECORD 256
TYPE ASCII
INTERACTIVE
DEBUG
STANDARDIZE us_addr
OUTFILE us_addr.txt
```

Parsing parameters

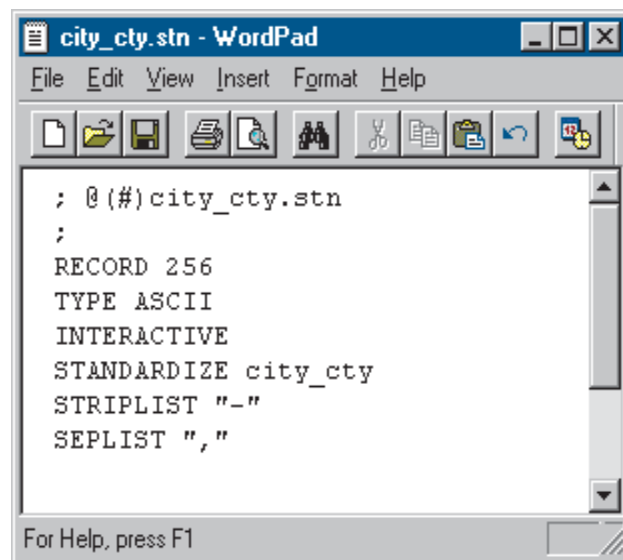
Parsing parameters (*SEPLIST* and *STRIPLIST*) are rules that define what constitutes a *token* or operand as defined in the pattern file. The *STRIPLIST* is a list of characters or symbols to be stripped during the standardization. By default, *STRIPLIST* includes: “,.\’;:”. The *SEPLIST* is a list of characters or symbols used to identify separate tokens. By default, *SEPLIST* includes “()-/,#&;:”. These default assumptions are overwritten by specifying a new *SEPLIST* and a *STRIPLIST*. If no new lists are specified, the default values are used.

Characters in the *STRIPLIST* or *SEPLIST* list must be enclosed in quotation marks. The quotation mark itself may not be in either list. Also, when you perform standardization, any character in the *STRIPLIST* is removed as if it never existed. For example, periods are stripped from the input records as if they never existed. Thus, N.W. becomes NW and is considered to be a single word.

Characters in the *SEPLIST* are used to separate tokens. Hyphens separate words, and each hyphen is considered a word in itself. For example, 123-456 is three words (tokens): 123, the hyphen (-), and 456.

Any character in both lists, such as a space, will separate two tokens but will not appear as a token. One or more spaces will be stripped, but the space indicates where one word ends and another begins. Spaces are both stripped and separate tokens. For example, “123 Main St” consists of three tokens: 123, Main, and St.

A *STRIPLIST* and a *SEPLIST* must immediately follow the *STANDARDIZE* statement to which they apply. The following example demonstrates how you might set up *SEPLIST* and *STRIPLIST* if you don’t want to use the defaults.

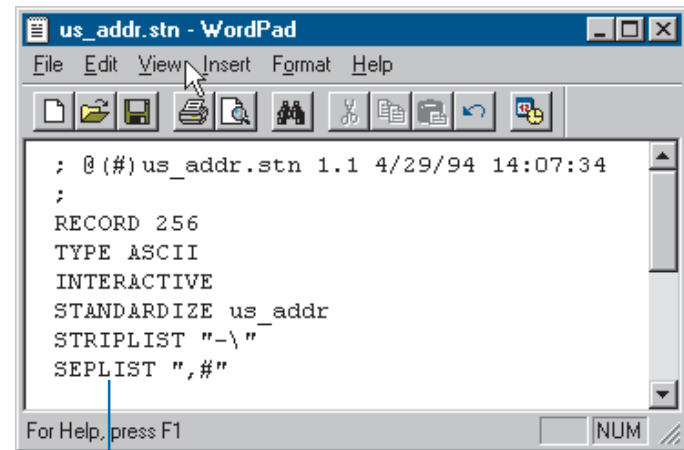
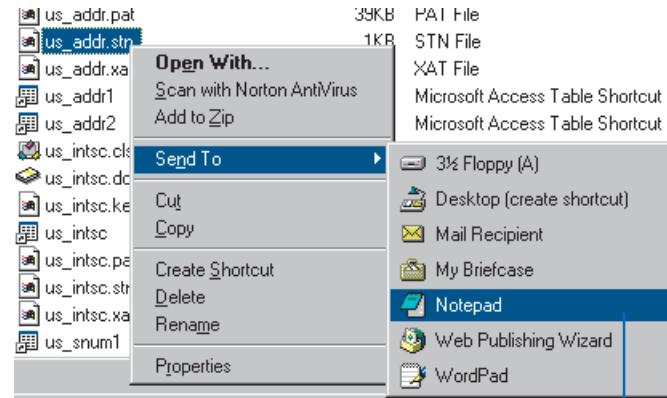


```
city_cty.stn - WordPad
File Edit View Insert Format Help
; @(#)city_cty.stn
;
RECORD 256
TYPE ASCII
INTERACTIVE
STANDARDIZE city_cty
STRIPLIST "-
SEPLIST ",
For Help, press F1
```

Adjusting parsing parameters

Parsing parameters are specified by default in the command file. However, in some cases, you may wish to change what characters are stripped or serve as spaces. In this case, you need to specify parsing parameters in your .stn file. If you want to accept the default, do not specify any parsing parameters. For a list of the default parsing parameters, see the section ‘Parsing parameters’ in this chapter.

1. Navigate to the .stn file you wish to modify.
2. Right-click the file, point to Send To, and click Notepad or another standard text editor.
3. After the STANDARDIZE command in the .stn file, add the parsing parameters.
4. Click the File menu and click Save.



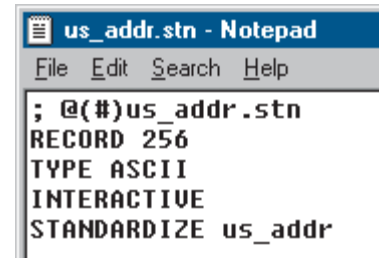
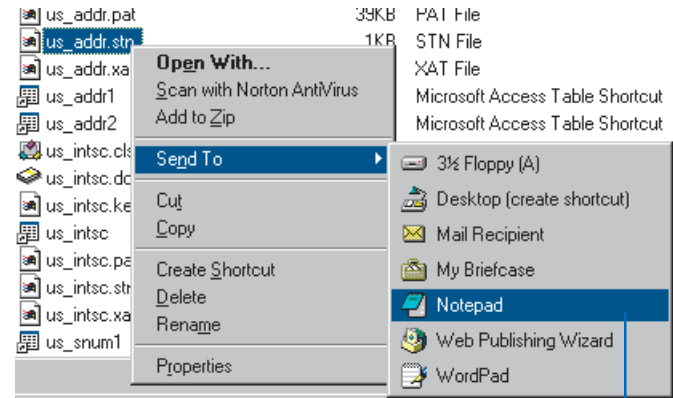
Adding the DEBUG and OUTFILE keywords to the us_addr.stn file

Before you can standardize and debug a process with *STANEDIT*, the software tool that tests and debugs the standardization rules, you must edit the standardization process *.stn file by adding the DEBUG and OUTFILE *keywords* to the command file.

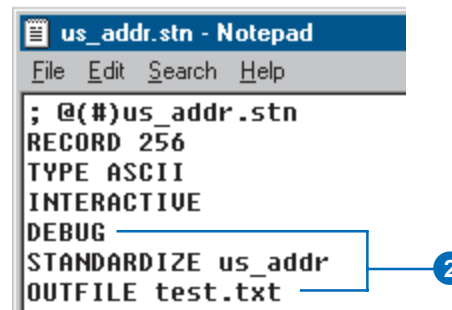
The DEBUG keyword puts the standardizer into debugging mode, and the OUTFILE keyword sets the debugging output to <file_name>. The filename can be anything.

Remember that you must remove the DEBUG and OUTFILE keywords prior to using the standardization process in the ESRI geocoding products.

1. Navigate to the us_addr.stn file in Windows Explorer. Right-click the file, point to Send To, and click Notepad. The us_addr.stn file displays.
2. Add the DEBUG and OUTFILE keywords to the file.
3. Click the File menu and click Save.



The us_addr.stn file is displayed.



The match key dictionary

5

IN THIS CHAPTER

- Overview of the match key dictionary
- Modifying the match key dictionary

The *match key dictionary* (.dct extension) is the file in the *standardization process* that defines information for the *match key field*. This chapter takes you through an overview of the match key dictionary, then shows you how it is formatted and how you can modify it.

Overview of the match key dictionary

The match key dictionary, which has a .dct extension, specifies the fields that an address may be parsed into as a result of the standardization. For example, a U.S. address contains the elements of house number, prefix direction, prefix type, street name, suffix type, and suffix direction. Thus, all these fields are defined in the match key dictionary file. Each field is called a match key. The match keys contain the values of the standardization result and are used for matching against the reference data.

The filename of the match key dictionary is always <file>.dct. Each line of the dictionary represents a field of the match key. The format is as follows:

```
<field-identifier> <field-type> <field-length> <missing value code> [; <comments>]
```

where

<field identifier>

The field identifier is a two-character unique field name (not case-sensitive).

<field type>

The field type defines how information is to be placed in the field.

C: character field; left-justified, filled with trailing blanks.

N: numeric field; right-justified, filled with leading blanks.

NS: numeric field; leading zeros are stripped.

M: mixed alphas and numerics; numeric values right-justified; alpha values left-justified in the fields. Leading zeros are retained if present. For example, 102 becomes b102, and A3 becomes A3bb, where b represents a space or blank.

MN: mixed name; field values beginning with a letter of the alphabet are left-justified. Field values beginning with a number are indented as if the number were a separate three-character field, for example:

```
MAIN
CHERRYHILL
bb2ND
b13TH
123RD
```

<field-length>

The <field-length> is the field length in characters.

<missing value code>

The <missing value code> is included for compatibility with the interactive matching library. The user should simply code an X for this operand.

X: no missing value

<comments>

Optional comments may follow a semicolon (;).

An example of this is:

```
\FORMAT\ SORT=N
```

HN	N	8	X; House number
PD	C	2	X; Predirection
PT	C	4	X; Pretype
SN	C	26	X; Street name
ST	C	4	X; Suffix type

SD	C	2	X; Suffix direction
XS	C	4	X; Soundex of street name
XR	C	4	X; Reverse Soundex of street name
ZN	C	20	X; Zone

The first line of a match key dictionary should always be
\FORMAT\ SORT=N. No comments can precede this line.

Modifying the match key dictionary

The match key dictionary (.dct file) is where the data type, field length, and missing value code for each *match key* field are defined. In some cases, you may want to add match key fields to the .dct file or remove ones that don't apply to your address data.

You can edit the match key dictionary in any standard text editor.

Remember that when you edit the .dct file, you need to update your .mat file accordingly.

When you update your .mat file, remove any of the match fields that you deleted in your .dct file. That's because if a field is deleted in the .dct file and is still in the .mat file, problems will crop up when the .mat file can't find the field to refer to it. Do not rename the match key names (that is, ►

Tip

Adding fields to the .dct file

When you add fields to the .dct file, you don't have to add them to the .mat file if you don't plan to create a match rule for the item you just added.

Adding match key fields to the .dct file

1. Open the us_addr.dct file in Notepad.
2. Add the fields you want to the file—in this case, City and State—and define the data type, field length, and missing value code.
3. Click the File menu and click Save when you are done with your edits.

```
\FORMAT\ SORT=N 1
; @(#)us_addr.dct
;
; Street address match key
;
HN   N   10   X; House Number
PD   C   2    X; Pre-direction
PT   C   6    X; Pre-type
SN   C   30   S; Street Name
ST   C   6    X; Suffix type
SD   C   2    X; Suffix direction

; Street address match key
;
HN   N   12   X; House Number
PD   C   4    X; Prefix direction
PT   C   8    X; Prefix type
SN   C   32   S; Street Name
ST   C   8    X; Suffix type
SD   C   4    X; Suffix direction
CT   C   28   X; City
SA   C   20   X; State 2
```

the field identifiers) unless you are sure that the match keys are not referred to by the pattern (.pat) or match rule (.mat) files. For more information on modifying your .mat file, see Chapter 3, ‘The match file’.

Tip

Removing fields from the .dct file

Although you won't usually remove fields from the .dct file, you may decide at some point that some fields aren't necessary for standardization or matching. At that point, you may decide to remove them.

Removing fields from the .dct file

1. Open the .dct file in Notepad.
2. Remove the field or fields that you don't want—in this case, Zone.
3. Click the File menu and click Save.

```
\FORMAT\ SORT=N  
; @(#) us_addr.dct  
;  
; Street address match key  
;  
HN N 10 X; House Number  
PD C 2 X; Pre-direction  
PT C 6 X; Pre-type  
SN C 30 S; Street Name  
ST C 6 X; Suffix type  
SD C 2 X; Suffix direction  
ZN C 20 X; Zone
```

```
\FORMAT\ SORT=N  
; @(#) us_addr.dct  
;  
; Street address match key  
;  
HN N 10 X; House Number  
PD C 2 X; Pre-direction  
PT C 6 X; Pre-type  
SN C 30 S; Street Name  
ST C 6 X; Suffix type  
SD C 2 X; Suffix direction
```

Zone has been removed from the .dct file.

The classification table

6

IN THIS CHAPTER

- Overview of the classification table
- How to get to the classification table
- How the classification table is formatted
- Modifying the .cls file

The *classification table* (.cls file) is one component in the files that is necessary to the *standardization process*. You can modify the .cls file so that it is more useful to you. For example, if you wish to change how words in your *reference file* are always standardized, you can edit the .cls file. This chapter describes the .cls file in detail and gives examples of various ways you can change it. The examples in the tasks come from the `us_addr.cls` file, which is used in several of the US Streets address styles. Because it's the most commonly used, it is a good example, but remember that with some of the address styles, it isn't used at all.

There are several other .cls files available, including:

- `us_intsc.cls`, used for intersection matching for the US Streets styles
- `zip.cls` and `zip4.cls`, used for ZIP5 style and ZIP5+4 style
- `stmap.cls`, used for StreetMap™ standard house address matching
- `key_1.cls`, used for the “Single Field” address style
- `city_cty.cls`, used for “City Country” address style
- `city_st.cls`, used for the “City State” address style
- `stm_int.cls`, used for StreetMap intersection matching
- `us_addrcls.cls`, used for the “US Alphanumeric Ranges” styles

Remember that not all ESRI products include all styles by default. For example, MapObjects only comes with `us_addr.cls`, `zip.cls`, and `zip4.cls`.

Overview of the classification table

The classification table (.cls extension) is used to identify and classify *keywords* that may appear in an address, such as street types (ST, AVE, BLVD) and directions (N, NW, S).

The filename of the classification table is always <file>.cls. The classification table is a standard ASCII file with one line per entry. Each entry contains:

```
<keyword>  
<standardized abbreviation>  
<keyword class>  
{<optional comparison threshold>}
```

where

<keyword> is the keyword that may appear in an address. It must be a single word.

<standardized abbreviation> is the abbreviation used to standardize various words of the same meaning. It would be classified into the appropriate field in the *match key*. Each abbreviation is limited to 25 characters in length.

<keyword class> is used in the *pattern files* to specify the rules that will be used to interpret the elements of an address. A *class* must be a single character.

Examples of a <keyword class> include:

0: Null—Of, the, for, 0 (null) is a special class.

D: Direction—East, West, Southwest

T: Street type—Avenue, Street, Place

M: Multiunit—Apt, #, Suite, Room

B: Box—P.O. Box

O: Ordinals—SECOND, THIRD

C: Cardinals—SIXTEEN, THREE

<comparison threshold> is the degree of uncertainty that can be tolerated in the spelling of the keyword, such as phonetic errors, random insertion, deletion, replacements, and transpositions of characters. The score is weighted by the length of the word, since small errors in long words are less serious than errors in short words. In fact, the threshold should be omitted for short words since errors in them generally cannot be tolerated. The numeric score operates roughly as follows:

900—exact match

800—strings are almost certainly the same

750—strings are probably the same

700—strings are probably different

An example of the classification table is:

EAST	E	D	
E	E	D	
NORTHWEST	NW	D	800.0
NW	NW	D	
AVENUE	AVE	T	800.0
AVE	AVE	T	
AV	AVE	T	
AVNUE	AVE	T	

How to get to the classification table

The classification table has a .cls extension. Depending on the application you're running, you can access the .cls file for editing from different places. MapObjects 2.x, ArcGIS, ArcView 3.x, and ArcIMS 4.x all store the .cls files in different places, but they all are similar in that all of the files are stored in one single flat directory with no tree below it.

If you are running MapObjects, the .cls files are stored by default in c:\program files\esri\mapobjects2\georules*.*.

If you are running ArcView 3.x, the .cls files are stored by default in c:\esri\av_gis30\arcview\geocode*.*.

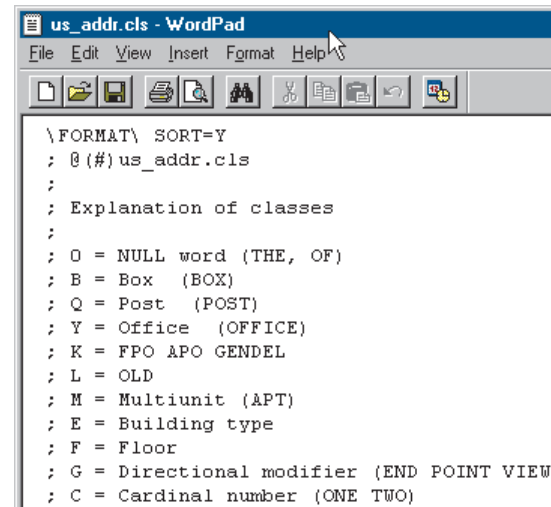
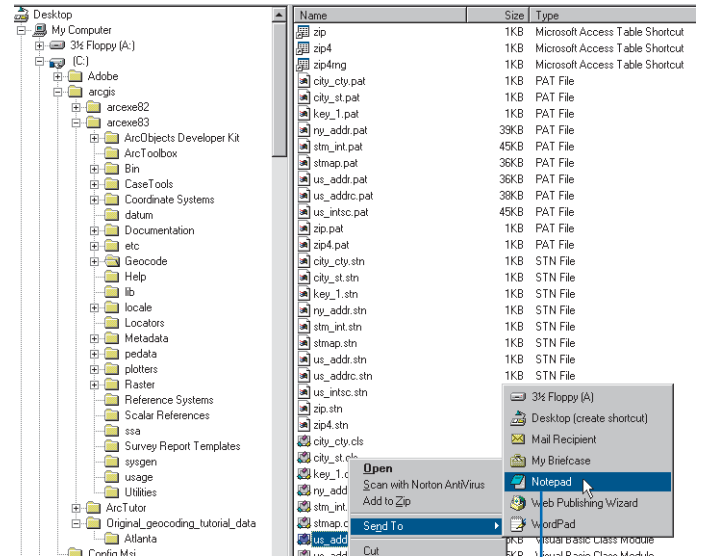
If you are running ArcGIS 9.0, the .cls files are stored by default in c:\arcgis\geocode*.*.

If you are running ArcIMS 4.x, the .cls files are stored by default in c:\arcgis\arcims\indexbuilder\styles*.* or c:\arcgis\arcims\server\ext\geocodeserver\styles*.*.

1. Locate the .cls file in Windows Explorer.
2. Open the file in a text editor, such as Notepad.

The file will display in Notepad.

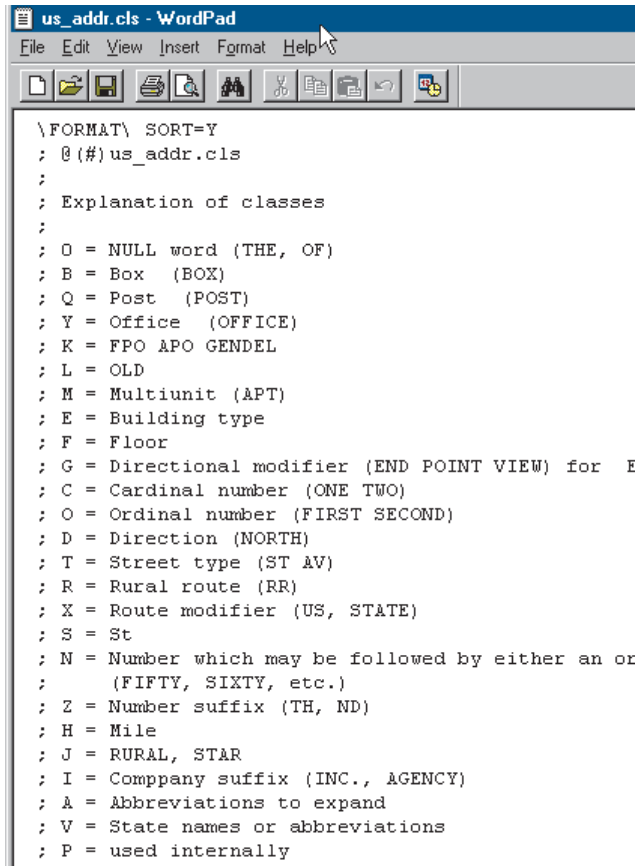
Note that if you would like to use another text editor, such as WordPad, you may do so, but make sure you don't save the file with any extra formatting.



The file displays in the text editor.

How the classification table is formatted

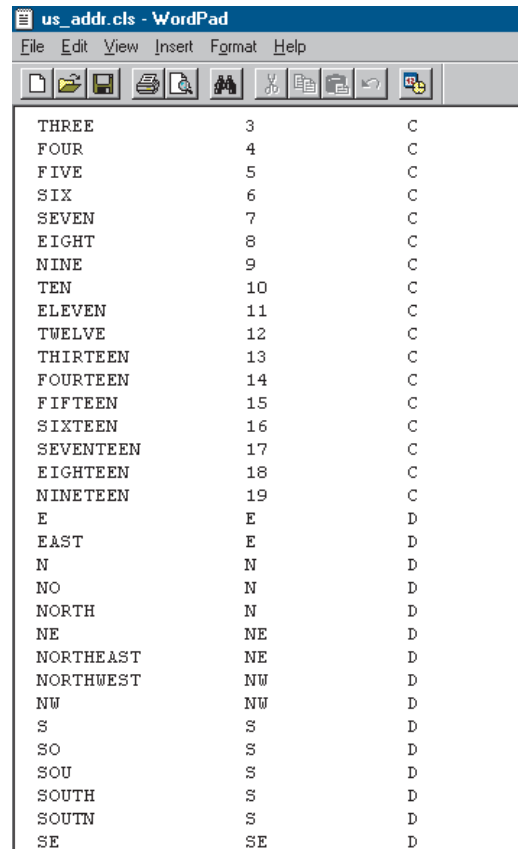
The classification table can be edited through Notepad or any other standard text editor. However, before you can begin to edit the .cls file, it is important to understand how the .cls file is formatted.



```
\FORMAT\ SORT=Y
; @{#}us_addr.cls
;
; Explanation of classes
;
; O = NULL word (THE, OF)
; B = Box (BOX)
; Q = Post (POST)
; Y = Office (OFFICE)
; K = FPO APO GENDEL
; L = OLD
; M = Multiunit (APT)
; E = Building type
; F = Floor
; G = Directional modifier (END POINT VIEW) for E
; C = Cardinal number (ONE TWO)
; O = Ordinal number (FIRST SECOND)
; D = Direction (NORTH)
; T = Street type (ST AV)
; R = Rural route (RR)
; X = Route modifier (US, STATE)
; S = St
; N = Number which may be followed by either an or
;   (FIFTY, SIXTY, etc.)
; Z = Number suffix (TH, ND)
; H = Mile
; J = RURAL, STAR
; I = Company suffix (INC., AGENCY)
; A = Abbreviations to expand
; V = State names or abbreviations
; P = used internally
```

Explanation of classes in .cls file

The .cls file consists of three columns; some rows also have an extra column with the comparison threshold. The first column is the keyword—that is, the information you may find in an address, such as EAST Avenue. All keywords in the file should be uppercased, and each keyword must be unique. Only one entry of



```
THREE 3 C
FOUR 4 C
FIVE 5 C
SIX 6 C
SEVEN 7 C
EIGHT 8 C
NINE 9 C
TEN 10 C
ELEVEN 11 C
TWELVE 12 C
THIRTEEN 13 C
FOURTEEN 14 C
FIFTEEN 15 C
SIXTEEN 16 C
SEVENTEEN 17 C
EIGHTEEN 18 C
NINETEEN 19 C
E E D
EAST E D
N N D
NO N D
NORTH N D
NE NE D
NORTHEAST NE D
NORTHWEST NW D
NW NW D
S S D
SO S D
SOU S D
SOUTH S D
SOUTH S D
SE SE D
```

Example of some columns as they appear in the classification table

the word is allowed in the table. Keywords can't contain spaces, numbers, or symbols. For example, "NORTHEAST" and "3RD" are not valid keywords. The second column is the standardized abbreviation. For example, if you enter the direction NORTH, it is standardized to N. The output standardized value can be anything you want, such as "3RD" or "MARTIN L KING". If the value contains more than one word, enclose the words in double quotes. All values should be uppercased. The third column is the class. An explanation of the classes occurs at the top of the .cls file. The optional fourth column is the comparison threshold, or the degree of uncertainty that can be tolerated in the spelling of the keyword.

Modifying the .cls file

The classification table takes a particular value in the address and assigns it a standardized abbreviation or value and a *token type value*. You can use the .cls file to add, remove, or modify street directions, types, and ordinal suffixes. For example, if you don't want "AVENUE" to be standardized as "AVE" because your reference data records "AV" as the street type, you may change it in the table. If you have some particular street types used in your databases, such as "CLOSE", that are not found in the existing table, you can add them to the table. For example, a new line like this:

```
CLOSE CLOSE T
```

can be added to the table.

When editing the .cls file, remember that any changes you make will be universal. If you wish to make a change in a special case (for example, filter for "North Bend" in a way that lets North be a part of the address for this address, but leaves it as a direction in all other cases), you can add a special routine to the .pat file. For more information, see Chapter 7, 'The pattern file'.

Modifying the .cls file to change standardization of ordinal suffixes

1. Open the .cls file in Notepad.
2. Scroll down to the ordinal numbers you wish to change, for example, FIRST.
3. If you wish to change your data so it is standardized as a full word (rather than the numeral), change the second column to the full word.
4. When you are done editing the .cls file, click the File menu and click Save.

FIRST	1ST
TWENTIETH	20TH
SECOND	2ND
THIRTIETH	30TH
THIRD	3RD

FIRST	FIRST
TWENTIETH	20TH
SECOND	2ND
THIRTIETH	30TH
THIRD	3RD

3

If your address data and reference data use different abbreviated values, you may need to modify how the term is abbreviated. The second column should match the format used in the reference material.

Modifying the .cls file to change AVE to AV

1. Open the .cls file in Notepad.
2. Scroll down to Avenue (use the Find tool in the Edit menu).
3. Change the text in the second column from AVE to AV.
4. Click the File menu and click Save.

AVENUE	AVE	T
AVNUE	AVE	T
	3	
AVENUE	AV	T
AVNUE	AV	T

In certain cases, some prefix values should not be removed from the name (for example, Calle Real) but are. This is because the rule base sees these words as prefix street types. You can fix this in the .cls file so these words are classified as part of the street name.

Removing Spanish street types for datasets that store prefix types in the street name field

1. Open the .cls file.
2. Scroll down to Avenida.
3. Comment out Avenida by placing a semicolon in front of it.
4. Repeat the process with Calle and Paseo, if desired.
5. Click the File menu and click Save.



```
AVENIDA AVE T
; AVENIDA AVE T
```


In some cases, you may have an unrecognizable street type. In a case like this, you can add the type to the .cls file, standardize the third column as a type, and save the .cls file. By doing this, you ensure that the words will be standardized correctly.

Adding new keywords and standard abbreviations to the .cls file

1. Open the .cls file in Notepad.
2. Scroll down to the bottom of the file.
3. Type the words you want to add (for example, Close) in the first column.
4. Type the words as they should be standardized in the second column.
5. Type a T in the third column to standardize as a type.
6. Click the File menu and click Save.

WAY
WY
CLOSE
3

WAY
WAY
CLOSE
4

T
T
T
5

When you are dealing with abbreviations for names of streets (for example, MLK or JFK), you may notice that all your *candidate* matching scores are low. This may be because the .cls file is standardizing the abbreviation to something other than what is in the reference data file. In other words, if the .cls file is standardizing MLK to MARTIN LUTHER KING, but the reference file contains MLK, the match score will be low because it won't be able to find the street name. To fix this problem, you can edit the .cls file so that MLK is standardized to MLK, and MARTIN LUTHER KING is standardized to MARTIN LUTHER KING.

Similarly, you may run into this sort of trouble when you are working with instances of ST. ST is a special case, since it can represent Street, Saint, the st in ordinal numbers (first, twenty-first, and so on), or Suite. The classification table is not the best place to deal with instances of ST, because it can't take different situations into account. Instead, it is handled in the pattern rules. For more information on dealing with instances of ST, see Chapter 7, 'The pattern file'.

Changing abbreviated names in the address data to match data in the reference files

1. Open the .cls file in Notepad.
2. Scroll down to MLK.
3. Change the second column from MARTIN LUTHER KING to MLK.
4. Click the File menu and click Save.

Your address data should now be the same as your reference file data, which will result in much better candidate scores.

RFK		"ROBERT F KENNEDY"	À
MLK	2	"MARTIN LUTHER KING"	À
KENNEDY		KENNEDY	À
KY		KEY	À
KYS		KEYS	À

RFK		"ROBERT F KENNEDY"	À
MLK	3	MLK	À
KENNEDY		KENNEDY	À
KY		KEY	À
KYS		KEYS	À

The pattern file

7

IN THIS CHAPTER

- Overview of the pattern file
- Pattern rules
- Actions
- Modifying the pattern file
- Dealing with street intersections
- Editing intersection .xat/.pat files
- Adding custom routines to the pattern file

The *pattern file* (.pat extension) is critical to the *standardization process* because it defines *pattern rules* and *actions*. This chapter looks at how the pattern file is set up, examines the different rules and actions that are available, and shows you how to modify the pattern file.

Overview of the pattern file

The pattern file (.pat extension) contains pattern rules and actions for standardizing an address and converting the recognized operands into *match key fields*.

The example below shows the three parts of the pattern file. The POST action section is optional and contains actions that are executed after patterns in the main section and subroutines are processed for the record. The pattern/action section shows that patterns and actions must be grouped together. This section can contain as many *pattern–action sequence* pairs as are necessary. The last section shows where *subroutines* are located.

```
\POST_START
<post-execution actions>
\POST_END
<pattern>
<action>
<pattern>
<action>
\SUB <action>
\END_SUB
```

The pattern file defines patterns to identify all elements of an address by keywords—for example, 123 North Main Avenue. This address can fit into the pattern:

Numeric : 123
Direction : North
Unknown word or words : Main
Street Type : Avenue

A pattern rule uses a pattern recognition syntax that will be discussed in detail in this chapter. A pattern rule is constructed in the form of a list of operands (that is, a representation of a token) separated by vertical bars. For example, a pattern of the above address would be

`^|D|?|T`

D and T classes are defined in the classification table. Each operand is referenced by an operand number. The above example contains four operands. The first operand is Operand [1], the second is Operand [2], and so on.

Actions defined after the pattern rule are executed to convert the value of the specified operand referred by its number to the match key fields. The match key fields were defined in the match key dictionary. The match key fields for the given example are:

Numeric : {HN}
Direction : {PD}
Unknown word or words : {SN}
Street Type : {ST}

Given the pattern, actions following the pattern rule for this address are:

```
COPY [1] {HN}  
COPY_A [2] {PD}  
COPY_S [3] {SN}  
COPY_A [4] {ST}
```

The first action copies the value in Operand 1 into the HN match key, that is, 123 in this example. The second action copies the abbreviated value of Operand 2 into the PD key, that is, “N”. Note that “N” was defined in the classification table as the standardized value for the keyword “NORTH”. The third action copies the entire string, including spaces in Operand 3, into the SN key, that is, “MAIN”. The last action copies the abbreviated value of Operand 4 into the ST key, that is, AVE as defined in the classification table for the keyword “AVENUE”.

To be correctly interpreted and moved to the match key fields, an address may require more than one pattern. Because of this, subroutines of pattern matching are allowed. An example of this comes when you need to define patterns to filter apartment units and standardize the rest for 123 N Main St Apt 101. To do this, you would use:

```
%1M ; M is a class for apartment units  
CALL APTS
```

```
^ | D | ? | T  
COPY [1] {HN}  
COPY_A [2] {PD}  
COPY [3] {SN}  
COPY [4] {ST}  
EXIT
```

```
\SUB APTS  
; patterns & actions for processing apartment  
units  
\END_SUB
```

The pattern file is a rule base file. Each rule is fired sequentially, unless it goes to a subroutine. The system will try to match the address to the first pattern. If it matches, the actions following the pattern will be executed. If it doesn’t match the pattern, it will proceed to the next pattern, and so on. If the matched pattern doesn’t end with an “EXIT” or “RETURN” action, it will continue to match to the next pattern. Subroutines can be introduced in the main section of the pattern file. Rules defined in the subroutines will be matched sequentially, unless a “RETURN” action is executed or when it reaches the end of the routine. After the subroutine is processed, the system will return to the previous location in the pattern–action sequence. When an EXIT action is executed or when it reaches the last pattern in the pattern file, the entire process will end.

The pattern file needs to be encrypted before it is installed. You can use the Standardizer Editor (*STANEDIT*) or the *encodpat.exe* program to encrypt the file. For more information on STANEDIT and *ENCODPAT*, see Chapter 8, ‘Developer’s Kit tools’.

Pattern rules

There are two types of pattern rules: unconditional and conditional. Unconditional patterns are strict rules that are not sensitive to the values of the operands. Conditional patterns allow patterns to match only under specified circumstances.

Unconditional patterns

The following are some simple pattern classes:

A–Z: Classes supplied by user from classification table

^: Numeric

?: One or more consecutive unknown alpha words

+: A single alphabetic word

&: A single token of any class

>: Leading numeric, for example, 3EBA

<: Leading alphabetic, for example, A501

@: Complex mixed, for example, 6H46K

-: Hyphen

The null class (0) is a special class that is used either in the classification table or in the RETYPE action to make a token null. Since it never matches to anything, it would never be used in a pattern. An example of using RETYPE to make the token null is:

```
-T | + | +      ; Avenue of America  
RETYPE [2] 0    ; removes the word "of"
```

The pattern represented by the address 123 N State St can be coded as:

```
^|D|?|T
```

A single alphabetic word can be matched with a + class. This is useful for separating the parts of an unknown string. For example, the pattern represented by Martin Luther King can be coded as:

```
+|+|+
```

As this pattern demonstrates, each of the words in Martin Luther King is tokenized into a +.

The pattern represented by 123A Main St can be coded as:

```
>|?|T
```

The > token matches to 123A (because a numeric is leading), the ? token matches to Main (the unknown), and the T token matches to the type.

The leading alphabetic class, which is represented by <, matches to patterns represented with a leading alphabetic character. For example, the pattern represented by ABC123 New York Ave would be coded as:

```
<|?|T
```

The complex class, which is represented by @, matches those tokens having a complex mixture of alphabets and numerics, for example, A123B. So, the pattern represented by A236C Crosier Blvd would be coded as:

```
@|?|T
```

Special single character classes

In some instances, a pattern may be represented by numeric or alphabetic characters that contain hyphens or slashes. The hyphen (-) and slash (/) are special single character classes.

Some examples of special single character classes are:

```
^|-|^      ; 123-127
```

```
^|^|/|^    ; 123 1/2
```

In these examples, the ^ represents a numeric, and the - or / operands represent the - or / in the pattern.

Single token

A *single token* of any class is represented by the ampersand (&).

An example of this is:

```
* M | & ;Apartment 3G, Room 45
```

End of field

The \$ *specifier* matches to the end of the field instead of matching to any real token. It is used to ensure that no tokens are left in the field after the pattern.

Here is an example of how the \$ specifier is used:

```
*^ | $ ;tests only 92373 of CA 92373
```

Subfield classes

The *subfield classes* (1 to 9, -1 to -9) are used to pick off individual words of a ? string.

1 = the first word

2 = the second word

-1 = the last word

-2 = the next to last word

If the referenced word doesn't exist, for example, ^|-|2 for 123-A where there is no second word in the column, the pattern doesn't match.

Subfield classes are useful for processing address suffixes, such as 123-A MAIN ST.

The pattern ^|-|1 would match as follows:

```
[1]= 123
```

```
[2]=-
```

```
[3]=A
```

Subfield ranges

Subfield ranges (beg:end) specify a range of unknown words:

(1:3) specifies words 1–3 of an unknown string.

(-3:-1) specifies the third-from-the-last to the last word of an unknown string.

(1:-1) specifies the first to last word of an unknown string (however, using ? is more efficient).

For example:

```
^ | - | (1:2) ;123 - A B Main St
```

```
COPY [3] {HS}
```

results in AB being moved to the {HS} match key field.

Universal class

The *universal class* (**) matches all tokens. For example, a pattern of ** would match 123 MAIN ST.

The universal class can be combined with other operands to restrict the tokens grabbed by the class. For example, the pattern 123 N Main St would be coded as:

```
** | T
```

The universal class deals with everything before the type, which may be no tokens. In this example, the universal class represents the ^ and ? tokens. In this case, Operand [1] = 123 N Main, and Operand [2] = St.

Floating positioning specifier

The *floating positioning specifier* (*) is used to modify the positioning of the pattern matching. The class immediately following it is a floating class. The pattern is searched until there is a match or the entire pattern is scanned. For example:

```
* M | ^ ;123 Main St Apt 34
```

Operand [1] = Apt (M is a class for Apt defined in the Classification Table)

Operand [2] = 34

This pattern can be used to filter out the apartment number information by retyping it to a null class. Essentially, it removes it from consideration by any patterns that appear further down the file of patterns.

It is important to remember that the asterisk must be followed by a class, for example, * M or * ^.

Reverse floating positioning specifier

The *reverse floating positioning specifier* (#) is similar to the floating positioning specifier (*), except that scanning proceeds from right to left instead of from left to right.

The specifier can only appear in the first operand of a pattern, since it is used to position the pattern. For example:

```
#S | ^ ;California 45 Products, Phoenix Arizona  
85042 (Class S for State names)
```

would scan from right to left for a state name followed by a number.

Operand [1] = Arizona

Operand [2] = 85042

Fixed position specifier

The *fixed position specifier* (%n) specifies the position at a particular operand in the input string.

%1 = the first token

%2 = the second token

%-1 = the last token

%-2 = the second-from-last token

You can qualify the positions by following the %n with a token type. Each token is treated according to its type:

%2^ = the second numeric token

%3T = the third type token

This specifier (%) is only permitted as the first operand of a pattern, since it is used to position the patterns.

For example, John Doe 123 Martin Luther St Salt Lake:

%1 1 matches to the John, the first word of the first string.

%3+ matches to the third single alpha word, Martin.

Negation class qualifier

The *negation class qualifier* (!) indicates NOT.

So, !T means not Type and will match to any token except a street type. For example, * M | !T matches to SUITE 3 but not to SUITE STREET.

The negation class may be combined with the floating class (*) only at the beginning of a pattern.

```
*! ? | T =T= "ST" | + ;ST CHARLES ST
```

```
RETYPE [2] ? "SAINT" ;expands ST to SAINT
```

This pattern matches to the string "ST CHARLES", and ST will be expanded to SAINT.

Conditional patterns

Conditional patterns allow patterns to match only under specified circumstances. For example, `T|?` matches to both `ST CHARLES` and `AVENUE OF THE AMERICAS`, where `ST` and `AVENUE` were defined as being in a `T` class in the classification table.

The actions required in both of these cases are different. In the first case, the `ST` refers to `SAINT`, and in the second case the `AVENUE` is the real street type.

Providing conditional values in patterns can correctly process such problem cases.

Simple conditional values

With *simple conditional values*, the operand is always followed by an equal sign and a value. The equality operator (`=`) tests both the standardized abbreviation and the original token value for equality to the operand.

When using simple conditional values, alphabetic values must always be in double quotes. However, numeric values are coded without quotes. An example of this is:

```
*T = "ST" | + ;ST. CHARLES  
RETYPE [1] ? "SAINT"
```

This is a street type whose value is `ST` followed by an unknown string.

The value to be compared is based on both the standard abbreviation—if one exists—and the complete word. This is useful because it prevents you from having to code all possible values.

```
D = "SOUTH" ; the word SOUTH is in the  
classification table.
```

```
D = "S" ; any direction with the standard  
abbreviation S.
```

Sometimes, operands are followed by `=A=` or `=T=`. In this case, `=A=` only tests the abbreviation, while `=T=` only tests the original token value. Some examples of this type of scenario are:

```
*T =T= "ST" | + ;test to see if ST was actually  
coded.
```

```
RETYPE [1] ? "SAINT"
```

```
D =A= "E" ;test the abbreviation regardless if  
the original word was EAST or E.
```

Series of conditional values

A *series of conditional values* is specified by delimiting the entries. The entries can be delimited using either spaces or commas. So, for example, the conditional values can be specified either by:

```
T = "ST", "AV", "PL"
```

or:

```
T=A= "ST" "AV" "PL"
```

A series of values can be tested against the *match key* in a similar fashion. For example:

```
^ | T | {SN} = "MAIN", "ELM", "COLLEGE"
```

If you are testing a matching key, the test must follow all pattern operands, including *end of field*.

Tables of conditional values

A large number of conditional values can be specified in additional tables. For example, you may want to use tables when you have titles for streets (such as `General John Doe`, `General J D`, and `General J Doe`), but all these titles refer to the same street.

The table parses the field to make sure it's part of the list in the table. You can specify tables as follows:

@<table file name>

The table is an ASCII file that has one line for each conditional value.

When you are testing a match key, the test against a table of values must follow all pattern operands, including an end-of-field operand.

For example:

^ | T | {SN} = @strtname.dat

means that the table strtname.dat is examined to check if the value in the SN key is found in the table.

Conditional expressions

A *conditional expression* is enclosed in square brackets immediately following the pattern operand. It can be either a <left-operand>, a <relational-operator>, or a <right-operand>.

Operands can be a variable name, the match key contents for the current operand, the match key contents for any field, a literal, a constant, or the special variables LEN and PICT.

Left operand in conditional expressions:

variable name

{}

{ } PICT

{ } LEN

{<match key name>} PICT

{<match key name>} LEN

variable name PICT

variable name LEN

{<match key name>}

<arithmetic-expression>

Relational operators in conditional expressions

<	original token is less than
>	original token is greater than
=	abbreviation or original token is equal to
=A=	abbreviation is equal to
=T=	original token is equal to
<=	original token is less than or equal to
>=	original token is greater than or equal to
!=	abbreviation and original token are not equal to
!=A=	abbreviation is not equal to
!=T=	original token is not equal to

Referencing current operand contents

You need to reference current operand contents when you need to reexamine special cases in the current operand.

?|S|^[{ } > 50000] ;zip code > 50000

The special operand { } indicates the contents of the current operand.

When character literals are in an equality test, the standard abbreviation is tested, if one is available. If this fails, the original input is tested. Some examples of this are:

T[{}= "ST"] ;tests the abbreviation because ST is listed in the classification table.

T[{}= "STREET"] ;tests the entire word STREET since it is not an abbreviation.

When comparisons other than the equality operators are specified, for example, T[{} <= "ST"], the original input is used rather than the abbreviation.

Referencing match key contents

When you are evaluating the match key for a particular instance only, such as {ZP} for ZIP Code, you are referencing match key contents. This specifies the match key field name enclosed in braces. For example:

```
^|D="E" |D="W" |T="HWY" | [{ZP}]>=20100&{ZP}<=20300]
;123 East West Highway, ZP=20200
```

Referencing constants, literals, and variables

You can reference constants, literals, and variables.

Numeric constants are referenced by coding a number. Negative numbers and decimal points are not permitted in numeric constants.

An example of a reference of a numeric constant is:

```
^ [{}=10000]
```

Literals are character constants. They are represented by enclosing a string in quotes.

A null or empty value is indicated by two consecutive quote marks:

```
? [{}=""]
```

Variables can be given any name desired. However, the name can't exceed 32 characters, and the first letter must be alphabetic. After the first character, any combination of alphabetic, digits, or underline characters may be used. A variable's type is numeric if it

is set to a numeric value. If it is set to a literal value, its type is character. For example:

```
[postcode = 12345] ;postcode is a variable.
```

Referencing the length of an operand

A LEN qualifier represents the length of an operand:

{ } LEN: Length of current operand

<variable> LEN: Length of contents of variable

{<match-key-field>} LEN: Length of match key field

For example:

```
^[{}LEN = 5] |-|^ [{}LEN = 4] ;search for a nine digit ZIP of the form 12345-6789.
```

If the numerics do not match the length of the operand, the pattern will not match. Some other examples of the LEN qualifier are:

```
?[temp LEN = 5] ;test the length of a variable.
```

```
[{SN} LEN = 20] ;test the length of a match key field.
```

Referencing a template of an operand

The PICT (picture) qualifier tests the special format of an operand:

```
<[{}PICT = "ccn"] |>[{}PICT = "ncc"] ;for testing English postal codes, e.g. AB3 5NW
```

The PICT defines how the numerics and alphabetic are distributed. ccn means character-character-number, and ncc means number-character-character.

Only the equality (=) and inequality (!=) operators may be used with PICT comparisons.

Referencing a substring of an operand

You can reference a substring of an operand to solve several situations. For example, this is useful when you are testing variables or match key fields. The following shows the format of referencing a substring of being an operand and some examples of how it would actually be used.

`{}` (beg:end): Substring of current operand

`<variable>` (beg:end): Substring of contents of variable

`{<match-key-field>}` (beg:end): Substring of match key field

where:

`(beg:end)` = the beginning and ending columns

`1` = the first column of the string

`-1` = the last column of the string

Some examples of this are:

```
+[{(-7:-1) = "STRASSE"}|^ ; HESSESTRASSE 15
```

```
[temp(2:4) = "bcd"] ;tests a variable
```

```
[{SN}(1:4) = "FORT"] ;tests a match key field
```

Separating spaces are removed by the parsers when testing substrings on multitoken values.

Arithmetic expressions

Arithmetic expressions can be `<left-arith-operand>`, `<arith-operator>`, or `<right-arith-operand>`.

A left-arithmetic-operand may be:

variable name

```
{ <match key name> }
```

```
{ }
```

Available arithmetic operators:

+ addition

- subtraction

* multiplication

/ division

% modulus

A right-arithmetic-operand may be:

variable name

constant

Arithmetic is limited to one operation per expression. No parentheses are permitted.

Examples:

```
^[{}/3 > temp] ;value in variable temp is 2 (in which case value in the ^ token should be greater than 6 in order to match this pattern)
```

```
^[{]%2 = 0] ;even numbered houses
```

Combining conditional expressions

Conditional expressions may be combined using the logical operators:

&: and |: or

All operations are executed left to right.

Some examples of combined conditional expressions include:

```
^[{]>=1000 & {]<=10000] ;house number in the range of 1000 to 10000
```

```
^[{]<100 | {]>1000] ;house number less than 100 or greater than 1000
```

Vertical lines within the square brackets are logical OR operations, and those outside the brackets are operand separators.

Actions

Whenever a pattern matches, a series of associated actions is performed. Actions can be used for many things, including filtering noise, converting an operand to other values based on a lookup table, copying a value or operand to a match key field or a variable, or invoking a subroutine.

Action commands must always be in uppercase.

Copying information (COPY)

The *COPY* command copies the entire string (all words are concatenated). There are no intervening spaces. The form of the *COPY* command is a <source> and a <target>.

The <source> can be an operand, substring operand, mixed operand, user variable, match key field, literal, or constant.

The <target> may be a match key field or user variable.

For example:

```
^ | ? | T ;123 MAIN ST
COPY [1] {HN}
COPY [2] {SN}
COPY_A [3] {ST}
EXIT
```

This pattern–action sequence will produce the result:

```
{HN} = 123
{SN} = MAIN
{ST} = ST
```

Copying substrings

A substring of an operand can be copied using the substring operand form. This form is:

```
COPY <source>(b:e)<target>
```

where b is the beginning letter of the string and e is the ending letter. For example:

```
+ | ^ ; HESSESTRASSE 15
COPY [1](1:5) {SN}
COPY [1](-7:-1) {ST}
COPY [2] {HN}
```

Copying leading/trailing characters

You can also copy leading or trailing characters. There are four possible mixed operand specifiers:

- (n) all leading numeric characters
- (-n) all trailing numeric characters
- (c) all leading alphabetic characters
- (-c) all trailing alphabetic characters

For example:

```
> | ? | T ;123A MAIN ST
COPY [1](n) {HN}
COPY [1](-c) {HS}
COPY [2] {SN}
COPY_A [3] {ST}
EXIT
```

Copying user variables

A user variable may be the target and/or the source of a *COPY*. The type of target user variable is determined by the type of the source. Some examples of this are:

```
COPY [1] temp ;copy an operand
COPY "SAINT" temp ;copy a literal
COPY temp1 temp2 ;copy a variable
```

Copying match key fields

You can copy match key fields to other match key fields. One example of this is:

```
COPY {HN} {HC}
```

In this example, the value of {HN} is copied to {HC}.

Copying standardized abbreviations (COPY_A)

You can copy the standardized abbreviation coded in the classification table for an operand to a target. The target can be a match key field or a user variable. For example:

```
^ | ? | T ;123 DAYTON AVENUE
```

```
COPY [1] {HN}
```

```
COPY [2] {SN}
```

```
COPY_A [3] {ST} ;copy AVE instead of AVENUE
```

Copying with spaces (COPY_S)

(*COPY_S*) preserves spaces between words when you copy them. An example of this is:

```
^ | ? | T ;123 OLD CHERRY HILL RD
```

```
COPY [1] {HN}
```

```
COPY_S [2] {SN} ;OLD CHERRY HILL
```

```
COPY_A [3] {ST}
```

Moving information

When you use move information (*MOVE*), the source is erased (made null) after it is moved. This action is similar to *COPY*, but *COPY* doesn't erase the source after the value is copied.

You can use this command to move a user variable or a match key field to another match key field. For example, you could do one of the following:

```
MOVE {HN} {HS}
```

```
MOVE temp {SN}
```

Concatenating information

You can concatenate information using either the *CONCAT* command or the *PREFIX* command. The *CONCAT* command concatenates information to a user variable or a match key field. The source can be an operand, a *literal*, or a user variable. For example:

```
^ | D="E" | D="W" | T="HWY" | [{ZP}]>=20100&{ZP}<=20300]
```

```
;123 East West Highway, ZP=20200
```

```
COPY [1] {HN} ; move house number to {HN}
```

```
COPY [2] temp
```

```
CONCAT " " temp
```

```
CONCAT [3] temp ;concat EAST WEST
```

```
COPY temp {SN} ;move to street name
```

```
COPY_A [4] {ST} ;move HWY to street type field
```

You can also concatenate the standard abbreviation of the operand using *CONCAT_A*.

The *PREFIX* command adds the concatenated operand to the beginning of a string.

PREFIX_A lets you prefix the standard abbreviation instead of the raw data. The source must be an operand.

Converting information (CONVERT)

CONVERT lets you convert data according to a lookup table or a user-supplied literal.

The following graphic is an example of the ASCII file codes.tbl:

001	"SILVER SPRING"
002	BURTONSVILLE 800.0
003	LAUREL

The first column above is the input value, and the second column is the replacement value. As you can see, multiple words are enclosed in double quotes.

Optional weights may follow the second operand to indicate that uncertainty comparisons may be used.

An example of how to use CONVERT is:

& ; any token

```
CONVERT [1] @codes.tbl TKN
```

TKN indicates that the token will be changed permanently for any pattern and/or action sets further down the current pattern file or for another process if that process uses the same token table. It will remain permanent until you finish the task. If you only want the conversion applied to the current set of actions, use TEMP instead.

Retyping operands (RETYPE)

RETYPE can be used for several different things. You can change the type of an operand in the token table, change the value of an operand, or change the abbreviation of an operand if it is found in the classification table.

When you use RETYPE, the format should be as follows:

```
RETYPE <operand> <class> [<variable> | <literal> ] [<variable> | <literal> ]
```

You can also retype an operand to a null (0) class to filter or remove the information. For example:

```
%M | & ;123 MAIN ST APT 56
```

```
COPY_A [1] {UT}
```

```
COPY [2] {UV}
```

```
RETYPE [1] 0 ;removes APT
```

```
RETYPE [2] 0 ;removes 56
```

If you want to change the token type and replace the text of a token, use RETYPE:

```
*!? | T = "ST" | +
```

```
RETYPE [2] ? "SAINT" ;changes the token type to ? and replaces ST to SAINT
```

When changing the standard abbreviation of the operand:

```
T=A="ST" | ^
```

```
RETYPE [1] M "SUITE" "STE" ;changes the standard abbreviation of SUITE to STE
```

Retyping multiple tokens (RETYPE)

When you are retyping multiple occurrences of a token, use <num>*<class> followed by a standard RETYPE statement referencing operand [1]. The number, which indicates the number of occurrences referenced, must be an integer from 0 to 255. Zero means that all occurrences should be scanned. An example of this is as follows:

```
0 * - ;123-45 MAIN ST
```

```
RETYPE [1] 0 ;removes all hyphens from tokens.
```

Soundex phonetic coding

Soundex codes are phonetic keys that are useful for blocking records in a matching operation. For more information about Soundex encoding, see Chapter 1, ‘Introduction’.

Soundex computes a Soundex code of a match key field and moves the results to the target match key field. The format should be as follows:

```
Soundex <source-field> <target-field>
```

For example:

```
Soundex {SN} {XS}
```

In addition to Soundex, you can also use reverse Soundex (RSOUNDEX). RSOUNDEX is useful for *blocking* on fields where the beginning characters might be in error. With reverse Soundex, the phonetic code is generated from the last nonblank character of the field and proceeds to the first.

These two actions are generally used in the POST action section so that they are executed after pattern matching is complete for the record.

Terminating pattern matching (EXIT)

When you are using *EXIT*, you can quit the pattern matching program for this process in this record. This prevents further pattern–action pairs from being executed.

The following example demonstrates how EXIT can be used:

```
^|?|T
```

```
COPY [1] {HN}
```

```
COPY_S [2] {SN}
```

```
COPY_A [3] {ST}
```

```
EXIT
```

Subroutines

Subroutines facilitate fast processing and can be invoked by a CALL action. Subroutine names should contain between 1 and 32 characters, the first character of which must be alphabetic. The format should be as follows:

```
CALL <subroutine>
```

For example:

```
%1 M
```

```
CALL APTS
```

Writing subroutines (SUB, END_SUB)

Subroutines always need to have a header and a trailer line:

```
\SUB <name>
```

```
    subroutine body
```

```
\END_SUB
```

Subroutines should always be coded at the end of the file. However, the order of the subroutines is unimportant. CALL actions are not permitted from within subroutines.

Control is returned back to the main program either when the \END_SUB is reached or when a *RETURN* action is executed. For example:

```
%1 M |^
```

```
CALL APT
```

```
\SUB APT
```

```
patterns and actions for processing apartments
```

```
\END_SUB
```


Returning from a subroutine (RETURN)

The RETURN action returns control from a subroutine to the main program. For example, in SUB_APTS:

```
M | ^  
RETYPE [1] 0  
RETYPE [2] 0  
RETURN
```

Modifying the pattern file

Some examples of things you can do in the pattern file include adding quickly filterable patterns, modifying existing rules to handle address pieces, adding custom routines, processing a specific case, and editing the intersection .xat/.pat files.

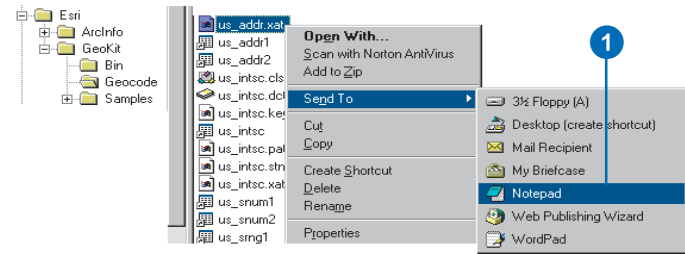
You don't make edits directly to the pattern file. Instead, you need to open the .xat file, which is the ASCII version of the file, make edits to it, then encode the .xat file to create a new .pat file. You can encode the .xat file using ENCODPAT.exe, one of the tools in the Geocoding Developer's Kit. For more information on ENCODPAT.exe, see Chapter 8, 'Developer's Kit tools'.

This section demonstrates some of the modifications you can make in the pattern file. They cover some of the most frequent issues that come up. You can also study them to see how the changes are made and apply what you see to your own situation.

Encoding the .xat file

1. Open the .xat file you wish to modify in Notepad. The .xat files are located in the geocode subfolder of the Geocode folder.
2. Make any changes in the .xat file.
3. Click the File menu and click Save.
4. Open STANEDIT.
5. Click the File menu and click Open Process.
6. Click the .stn file you wish to use (has the same filename as the .xat file you modified).
7. Click the Tools menu and click ENCODE PAT.

ENCODE PAT converts the .xat file into a binary file with the .pat file extension.



You can add quickly filterable patterns to the top of the .xat file, then EXIT. This lets you easily modify patterns. This task uses the example of the address 123 23rd St. Ct. as an example.

Adding quickly filterable patterns to the pattern file

1. Note in the address:
123 23rd St. Ct.
that 23rd St. represents the street name, and Ct.
represents the street type.
2. Open the us_addr.xat file in Notepad.
3. Add the following pattern at the top of the .xat file:

Pattern is:

```
^ | & | S | T ;  
standard form of 123 23rd  
St. Ct.
```

```
COPY 1 {HN}
```

```
COPY 2 temp
```

```
CONCAT temp
```

```
CONCAT_A [3] temp
```

```
COPY temp {SN}
```

```
COPY_A [4] {ST}
```

```
EXIT
```

4. Click the File menu and click Save.
5. Open STANEDIT (for information on using STANEDIT, see Chapter 8, 'Developer's Kit tools').
6. Click the Tools menu and click ENCODE PAT.

The .xat file is converted to a binary .pat file.

Sometimes, you may work with a particular address that doesn't follow the standard rules. For example, in the case of the address 235 Trail West Street, West is part of the street name. In this instance, you need to modify the .xat file so West is included as a street name for this instance but remains a direction for all other instances.

Modifying the .pat file for a special circumstance

1. Open the us_addr.xat file in Notepad.
2. To quickly process the special case for "Trail West", add a section to the top of the .xat file with the pattern outlined in the graphic to the right.
3. Click the File menu and click Save.
4. Open STANEDIT.
5. Click the Tools menu and click ENCODE PAT.

The .xat file will be converted to a binary .pat file.

2

```
;  
;  
;   Quickly process SPECIAL CASES  
;  
;  
;  
^ | T="TRAIL" | D="WEST" | $  
; example: 123 TRAIL WEST  
; "Trail West" is the street name.|  
COPY [1] {HN}  
COPY [2] temp  
CONCAT " " temp  
CONCAT [3] temp  
COPY temp {SN}  
EXIT
```

The classification of ST is often tricky. ST can stand for Street, Saint, ordinal numbers (for example, 1ST), or Suite and is *tokenized* and classified differently for each one.

The us_addr.pat file has been set up to handle the various meanings of ST. If you are setting up a new process and will need to handle ST, you can use the us_addr file as a guide.

Dealing with ST

1. Navigate to the .xat file you wish to modify (default location is C:\programfiles\ESRI\GeoKit\Geocode).
2. Right-click the .xat file, point to Send To, and click Notepad (or another standard text editor).
3. Add the text in the example at the right to the top of your .xat file.
4. Add the text in the example at the right to your .xat file after your pattern-action sequence pairs.
5. Click the File menu and click Save.
6. Open STANEDIT (default location is C:\program files\ESRI\Geokit\Bin).
7. Click the File menu, click Open Process, and click the process you are working on (same filename as the .xat file you modified).
8. Type an address to be standardized in the text box.
9. Click the Tools menu and click Encode PAT.

The .pat file is outputted to your geocode folder where the software is installed (for example, if you are using ArcGIS, the default location is C:\arcgis\arcexe83\geocode).

```

;-----
; Handle various meanings of ST: 1st, Saint, state, street, suite
;-----
*%S
CALL ST_PROCESS 3
;

;-----
; Handle various meanings of ST: 1st, Saint, state, street, suite
;-----

\SUB ST_PROCESS
*! ? | S | + [{} LEN > 2]
RETYPE [2] ? "SAINT"

;#^=11,111,12,112 | S ; 11 st
;COPY [1] temp
;CONCAT "TH" temp
;RETYPE [1] ? temp
;RETYPE [2] T

;#^ [ {} % 10 = 1] | S ; 1 st
;COPY [1] temp
;CONCAT_A [2] temp
;RETYPE [1] ? temp
;RETYPE [2] 0

%1S | T=A="RD" | ^ ; ST RD 34
RETYPE [1] X

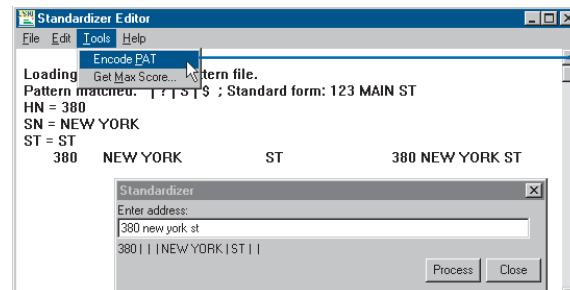
%1S | ? | T ; St Helens Pl
RETYPE [1] ? "SAINT"

%1S | ? | S ; St Helens St
RETYPE [1] ? "SAINT"
RETYPE [3] T "ST"

%1S | S ; ST 56 (abbrev for suite) --- make sure this is ok
RETYPE [2] M "STE" "STE"

%1S
RETYPE [1] T "ST"
\END_SUB
;

```



Dealing with street intersections

An address style can contain two matching processes:

- Primary street address, for example, `us_addr1.mat`
- Street intersection, for example, `us_intsc.mat`

Each of these processes has its own standardization process.

To ensure that the street name field in the reference file is standardized, an additional *standardization process*, `stname.stn`, can be added to the address style.

The street *intersection* and *reference file* street name standardization and matching processes are optional.

Matching and standardizing street intersections

An address style can contain a standardization and matching process for street intersections in addition to the street address standardization and matching process.

Street intersection matching only works on line (arc) feature types. By default, a street intersection is delimited by an ampersand (&). It doesn't contain a house number. An example of a street intersection is:

N State Street & Main Avenue NE

Defining MatchRules for street intersections

When you are defining MatchRules for street intersections, two sets of match variables for the components preceding and succeeding the delimiter should be defined. For example:

```
VAR PreDir1      1      2 X ; Prefix direction 1
VAR PreType1     3      4 X ; Prefix street type 1
VAR StreetName  17     28 S ; Street name 1
VAR Type1       35     6 X ; Suffix street type 1
VAR SufDir1     41     2 X ; Suffix direction 1
```

```
VAR PreDir2     43     2 X ; Prefix direction 2
VAR PreType2    45     4 X ; Prefix street type 2
VAR StreetName249 28 S ; Street name 2
VAR Type2       77     6 X ; Suffix street type 2
VAR SufDir2     83     2 X ; Suffix direction 2
```

In addition to the two sets of match variables, two sets of match rules are also needed:

```
MATCH UNCERT S1 StreetName1 0.9 0.01 700.0
MATCH UNCERT S2 StreetName2 0.9 0.01 700.0
MATCH CHAR P1 PreDir1 0.8 0.1
MATCH CHAR P2 PreDir2 0.8 0.1
MATCH CHAR E1 PreType1 0.7 0.1
MATCH CHAR E2 PreType2 0.7 0.1
MATCH CHAR T1 Type1 0.85 0.1
MATCH CHAR T2 Type2 0.85 0.1
MATCH CHAR D1 SufDir1 0.85 0.1
MATCH CHAR D2 SufDir2 0.85 0.1
```

Writing pattern rules for street intersection

Use an ampersand (&) as a delimiter in the pattern file to differentiate the two street segments that form the intersection. The & is a reserved character in a pattern file. To identify the & symbol from a text string, it must be preceded by a backslash (\), that is, `\&`.

There are some general procedures you should follow to write the standardization patterns for intersections:

1. Check if there is any illegal input. Exit if the input string contains more than one delimiter.

```
*\& | *\&
```

```
EXIT
```

2. Process the first half of an intersection by including the ampersand delimiter at the end of each pattern and patterns in the subroutines. For example:

```
? | T | *\&
```

3. Immediately retype the tokens to null and remove the \& delimiter after standardization of the first half of the intersection is complete. Use the following pattern right before standardization of the second half starts:

```
; remove the delimiter and everything
```

```
; before the delimiter
```

```
** | \&
```

```
RETYPE [1] 0
```

```
RETYPE [2] 0
```

4. Process the second half of the intersection. Only tokens for the second half are left. A pattern can be as simple as:

```
? | T | D
```

Editing intersection .xat/.pat files

When you need to modify a pattern file that contains intersections, you need to write an intersection process (you can't simply append to another process, for example, `us_addr`). The process `us_intsc.dct` exists for intersections. This task takes you through how to set up a process for intersections so you can do it if you need to create a new process.

The task uses the example of North Main Avenue & East Clark Boulevard. North Main Avenue (standardized to `N MAIN AVE`) is the first part, the `&` is the delimiter, and East Clark Boulevard (standardized to `E CLARK BLVD`) is the second part.

The address gets parsed by the `.dct` file into separate components. After it is parsed, you can write the pattern-action in the `.xat` file, then convert the file into the `.pat` file.

1. Look at the graphic on the right to see how the `.dct` file parses the address into components.
2. Navigate to the `.xat` file you wish to modify in Windows Explorer (the default location is at `C:\program files\ESRI\GeoKit\Geocode`).
3. Right-click the file, point to Send To, and click Notepad.
4. Type the pattern at the right in the `.xat` file. Note that `\&` denotes you are looking at an actual ampersand in the address, rather than an operand.
5. In the very last line (after `EXIT`), add a carriage return.
6. Click the File menu and click Save.
7. Open `STANEDIT` (default location is `C:\program files\Escri\GeoKit\Bin`).
8. Type an address to be standardized in the text box.
9. Click the Tools menu and click Encode `PAT`.

The `.pat` file is outputted to your geocode folder where the software is installed (for example, `C:\arccgis\arcexe83\geocode`).

P1	S1	T1	P2	S2	T2
N	Main	Ave	E	Clark	Blvd

1

```
D | ? | T | \&  
COPY_A [1] {P1}  
COPY_S [2] {S1}  
COPY_A [3] {T1}  
RETYPE [1] Ø  
RETYPE [2] Ø  
RETYPE [3] Ø
```

; type the following pattern under all the
; pattern-action sequences—this ensures that
; everything before the second part (`E CLARK
; BLVD`) is gone

```
** | \&  
RETYPE [1] Ø  
D | ? | T  
COPY_A [1] {P2}  
COPY_S [2] {S2}  
COPY_A [3] {T2}  
EXIT
```

4

Adding custom routines to the pattern file

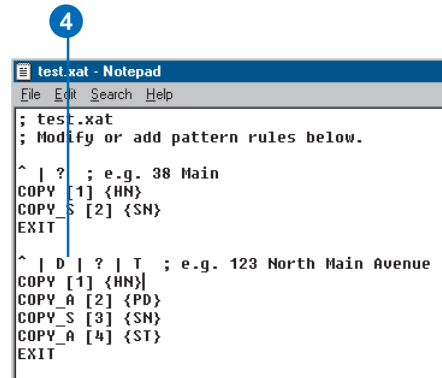
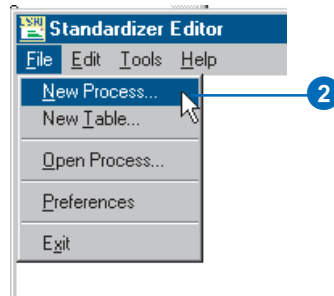
In some cases, you may want to add custom routines to the pattern file to handle special cases you may come across. This section looks at how you can do this.

Tip

Actions in the .xat file

Remember that all the actions in the .xat file must be uppercase.

1. Open STANEDIT (default location is C:\program files\ESRI\GeoKit\Bin).
2. Click the File menu and click New Process.
3. In the Save As dialog box, type the name of your new process, then click Save.
The files that make up the standardization process open.
4. In the .xat file, add the pattern–action sequence to standardize your address.
5. You can also add a custom routine, for example, to remove directions. This is useful if you want to standardize the address without worrying about the {PD} field. ▶



Tip

Creating a new process

To help everything go smoothly when you are creating a new process, you should first update your .dct file, then your .cls file, then your .xat file, then add the DEBUG and OUTFILE keywords to your .stn file. For information on adding the DEBUG and OUTFILE keywords to the .stn file, see Chapter 4, 'The command file'.

Tip

Editing an existing process to create a new process

If you wish, you can refer to an existing process, such as us_addr.*, and edit the files accordingly. An unencoded version of us_addr.pat is included in your ..\GeoKit\Geocode folder.

6. Type the text in the graphic at the right into the .xat file. DIRECTIONS is the name of the subroutine.

RETURN in the pattern–action sequence at the right is what moves the action to the next pattern.

A [2] is used under COPY_A [2] under \SUB_DIRECTIONS because the subroutine only deals with direction; this removes direction to streamline standardization.

7. When you are finished, click the File menu and click Save.
8. To create a new process, you need to add PD and ST to the .dct file. Add the text at the right to the .dct file you just created. For information on adding fields to the .dct file, see Chapter 5, 'The match key dictionary'.
9. You need to update the .cls file. Add the text at the right to the .cls file you just created. It should replace the "0 0 0" line. For information on adding fields to the .cls file, see Chapter 6, 'The classification table'.
10. Open the process in STANEDIT and enter an address to standardize.
11. Click the Tools menu and click Encode PAT to convert the .xat file into a .pat file.

```
6
test.xat - Notepad
File Edit Search Help
; test.xat
; Modify or add pattern rules below.
^ | ? ; e.g. 38 Main
COPY [1] {HN}
COPY_S [2] {SN}
EXIT
* | D ; wildcard
CALL DIRECTIONS
^ | ? | T ; Direction has been removed because of subroutine
COPY [1] {HN}
COPY_S [2] {SN}
COPY_A [4] {ST}
EXIT
\SUB DIRECTIONS
^ | D | ? | T
COPY_A [2] {PD}
RETYPE 2 0
RETURN
\END_SUB
```

```
8
PD C 2 X; Pre-direction
ST C 6 X; Suffix type
```

```
9
;
N NORTH D
NORTH NORTH D
AVENUE AVE T
AV AVE T
AVE AVE T
```

Developer's Kit tools

8

IN THIS CHAPTER

- **STANEDIT and the DEBUG and OUTFILE keywords**
- **Creating a new process with STANEDIT**
- **ENCODPAT**
- **What to do before adding your files to the folder**

The Geocoding Developer's Kit includes several tools that allow you to modify the *rule base* and *match files*. The main tool, *STANEDIT*, is the Windows version of Interactive Standardizer. It is used for *pattern rules* syntax checking and debugging. It executes under all Windows® operating systems, including Windows 95/98, Windows NT®, Windows 2000, and so on. *STANEDIT* includes *ENCODPAT.exe*, which can also execute under DOS. *ENCODPAT* is a pattern rule encryption program that is used to encode *standardization* pattern rules files for use in ESRI's geocoding products. Using *ENCODPAT.exe*, you can convert the ASCII *.xat* file into the binary *(.pat)* pattern file. *GETMAXSCORE* and *GEOPUB30.dll* are also included with *STANEDIT*. These programs are used in ArcView 3.x only—the other ESRI geocoding products, such as MapObjects, ArcIMS, and ArcGIS, perform these programs' functions internally.

You can run *STANEDIT* and its tools either from the interface or from the command prompt. This chapter covers how to execute these programs both ways, goes into detail about the tools in the Geocoding Developer's Kit, and illustrates how to use them in your applications.

STANEDIT and the DEBUG and OUTFILE keywords

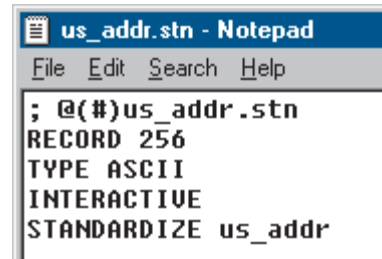
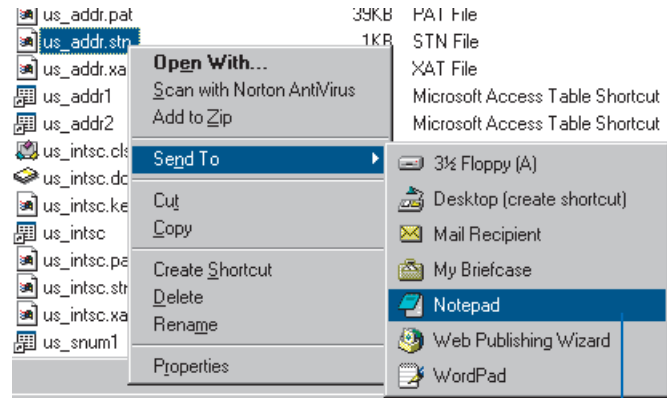
Before you can standardize and debug a process with STANEDIT, you must edit the *standardization process *.stn* file by adding the *DEBUG* and *OUTFILE* keywords to the *command file*.

The *DEBUG* keyword puts the standardizer into debugging mode, and the *OUTFILE* keyword sets the debugging output to <file_name>. The filename can be anything.

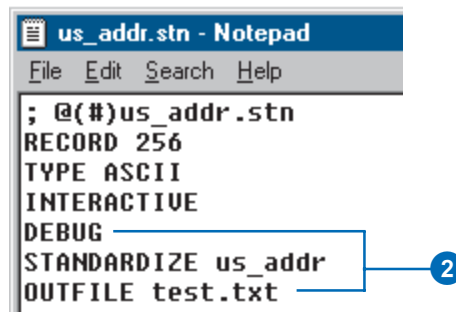
Remember that you must remove the *DEBUG* and *OUTFILE* keywords prior to using the standardization process in the ESRI geocoding products.

Adding the DEBUG and OUTFILE keywords to the command file

1. Open the *us_addr.stn* file in a standard text editor, such as Notepad.
The *us_addr.stn* file displays.
2. Add the *DEBUG* and *OUTFILE* keywords to the file.
3. Click the File menu and click Save.



The *us_addr.stn* file is displayed.



Once you have added the DEBUG and OUTFILE keywords to the .stn file, you are ready to standardize the address using STANEDIT. The patterns used for standardizing the input address are displayed on the main window. The output file specified by the OUTFILE keyword also contains the patterns that the input address was standardized against. You can examine the output file after the process is closed. The file will be overwritten every time the process is opened.

Tip

Entering multiple addresses

You can enter multiple addresses one after another in the Standardizer text box.

Tip

Checking pattern rules syntax using STANEDIT

STANEDIT also performs pattern rules syntax checking. STANEDIT will close the process after you click OK on the Error message box. To open the problem file and correct the errors, click the Edit menu and click the Last File option. The Last File option opens the file where the error occurs in the last edited file. Then, click the File menu and click Open Process to open the process again.

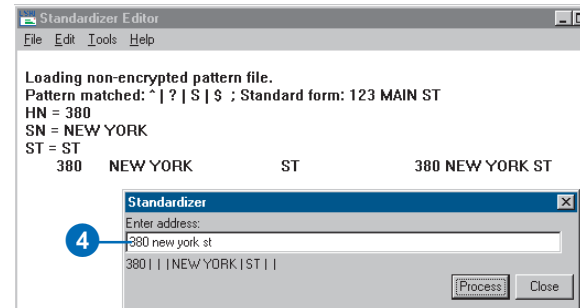
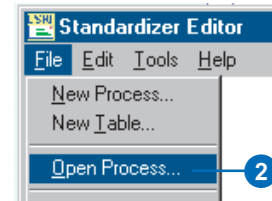
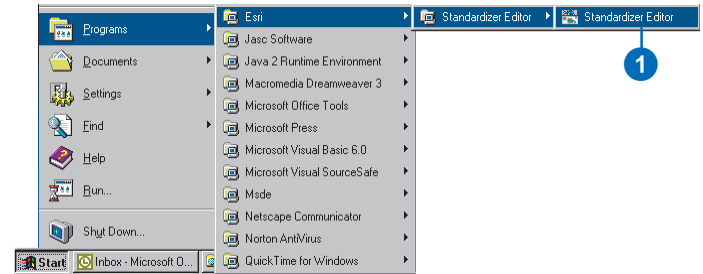
Using STANEDIT to standardize an address using us_addr.stn

1. Click Start, point to Programs, point to Esri, point to Standardizer Editor, and click Standardizer Editor.
2. Click File and click Open Process in the Standardizer Editor window.

The Open dialog box appears.

3. Click the us_addr.stn file in the Open dialog box.
4. Type the address you wish to standardize in the Standardizer text box and click Process.

The address will standardize in the Standardizer Editor window.



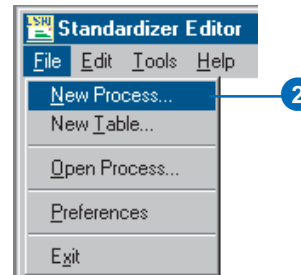
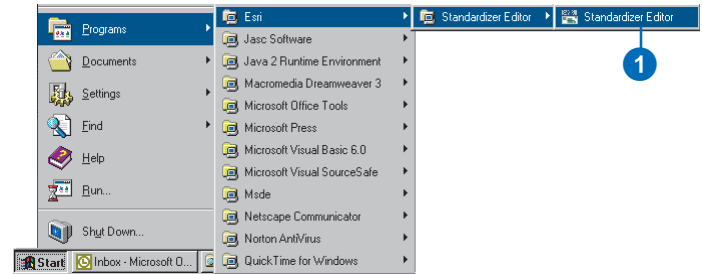
Creating a new process with STANEDIT

In some cases, you may not want to use any of the existing rule bases that come with your Geocoding Developer's Kit. In this case, you can create a new rule base. You may decide to do this when you need to work with *address formats* that aren't at all similar to the supported address formats provided with ESRI GIS products. For example, the `us_addr` address style works for American addresses, but perhaps not for other countries. If you are working with addresses that don't fit into the `us_addr` address style, you may want to create a completely new style.

1. Click Start, point to Programs, point to Esri, point to Standardizer Editor, and click Standardizer Editor.
2. Click the File menu and click New Process.
3. In the Save As dialog box, type the name of the new process, then click Save.

The files open in a text editor (the default is Notepad).

4. Edit the files and add the rules accordingly (for information on editing the files, see the section, 'Adding custom routines to the pattern file' in Chapter 7).
5. After you have edited each file, click the File menu and click Save.



ENCODPAT

When you have finished standardizing and debugging your files with STANEDIT, you can use ENCODPAT to encode the .xat file into a binary .pat file. The ESRI geocoding products require encoded pattern rule files—if you try to deploy unencoded pattern rule files with your applications, you may get unpredictable results and errors.

Remember that ENCODPAT is a one-way file encryption program. It can't be used to unencode files.

Tip

Running ENCODPAT from DOS

You can run ENCODPAT without STANEDIT. The command line at the DOS prompt is:

```
C:\geokit\geocode>..\bin\  
encodpat  
encodpat <in_file> <out_file>
```

For example:

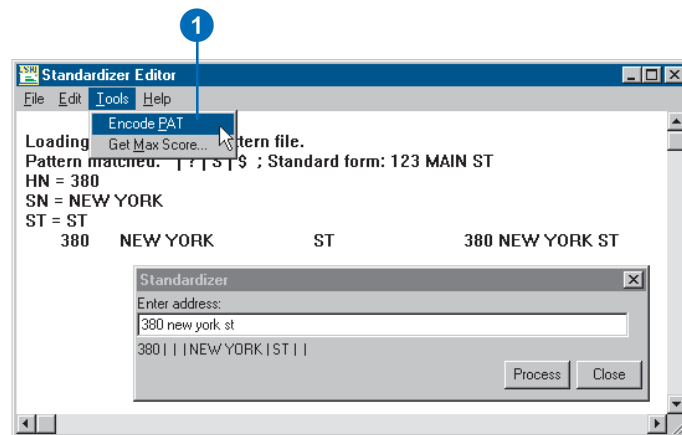
```
C:\geokit\geocode>..\bin\  
encodpat  
us_addr.xat us_addr.pat
```

Encoding the .xat file using ENCODPAT

1. When you are finished standardizing and debugging your process in STANEDIT, click the Tools menu and click Encode PAT.

If you are running ENCODPAT from STANEDIT, the .pat file will be outputted to the same folder where all the other files in the standardization process are.

If you are running ENCODPAT from the DOS prompt, the .pat file will go to the current folder of the prompt if you don't specify a path for the .pat file. Otherwise, everything will be saved to whatever folder it shows after the prompt.



What to do before adding your files to the folder

After you have performed a sufficient amount of testing and are sure that all the addresses you enter are standardized properly, there are a few more things to take care of before you can add the files back into your folder. This section provides a checklist for you to go through to make sure that your files are ready to be added to the folder.

1. In the .xat file, make sure that there is a carriage return after the last EXIT command.
2. The .xat file must be converted into a .pat file using ENCODPAT.
3. After running STANEDIT, remove the OUTFILE and DEBUG keywords from the .stn file.
4. Save the command file without the OUTFILE and DEBUG keywords.

After you have completed these tasks, you can move your files into the folder where your standardization process files and match files are stored.

Data dictionaries and match rules syntax

A

IN THIS APPENDIX

- **Introduction**
- **Data dictionaries and match specifications**
- **Geocoding**
- **Record linkage concepts**

This appendix contains the documentation about how to define data dictionaries and match rules used in geocoding as well as a section describing record linkage concepts.

The text in this appendix has been adapted from the MatchWare Technologies, Inc., AUTOMATCH User Manual (version 2.9). AUTOMATCH is a general purpose record linkage system that has been adapted for use in geocoding. With permission from MatchWare Technologies, this section has been modified by ESRI to make the described commands and file preparation specific to ESRI geocoding products. As a result, the discussion in this section does not necessarily represent the complete and original usage of AUTOMATCH.

Introduction

This appendix is divided into four sections—the first three sections describe the geocoding service for ESRI applications, while the fourth section presents a tutorial on record linkage concepts.

File preparation

ESRI geocoding products read the records of a theme's attribute table. The theme's attribute table becomes the file to be matched.

Field preparation

The actual data to be matched should conform to the following practices:

Each matching component should be stored in a separate field in the theme's attribute table.

Missing values should be distinguishable from zero values for numeric fields. Care should be taken about distinguishing a value of zero from a value-missing condition. Missing values are discussed in the next section.

Street addresses should be separated into individual standardized components. These components include house number, street directionals (NORTH, WEST), street names, street types (AVE, ST, BLVD), city name, state name, and postal code. Such standardization is required since matching complex character strings (such as 123 1 ST ST ST 56, which means 123 FIRST STREET SUITE 56) does not provide adequate results without sophisticated pattern recognition programs.

Dates should be in year-month-day order, if possible (for example, 19920304 = March 4, 1992), since this places data fields in ascending collating sequence.

Time should be represented as hour-minute (or hour)—for example, 1230 is 12:30 PM. Remember that 0000 is a valid time (Midnight).

Data dictionaries and match specifications

Each file that is to be a participant in a match requires a data dictionary associated with match specifications. The name given to the match specification file is the name of the entire matching application, with a .mat extension.

You can create the file directly with any standard text editor, such as the MS-DOS **EDIT** program. Don't use word processing programs to create these files, since they insert special control information for formatting and may not work. The 'Geocoding' section in this appendix presents the format of each statement.

The file must be a standard ASCII file created with a text editor. Each line represents one statement. A statement consists of a command followed by zero or more operands.

A data dictionary consists of a number of VAR commands. After the data dictionaries are defined, the user must define the matching specifications. It contains a number of commands. A command consists of a command name and a series of operands separated by one or more spaces.

Comments may be specified on a command line. All text following a semicolon (;) is considered to be a comment.

VAR command

The VAR command is used to define all of the variables (fields) on the file that may be useful for matching. A maximum of 100 variables may be defined for each dictionary. The VAR command has the following format:

VAR

<variable-name>

<beginning-column>

<length>

<missing-value code>

where <missing-value code> ::=

S: spaces

Z: zero or spaces

N: negative number (for example, -1)

9: all nines (for example, 9999)

X: no missing value

For example:

VAR STREET_NAME 1 20 X: the variable **STREET_NAME** begins in column 1 for length 20 and there are no missing values.

VAR STREET_TYPE 21 10 S: the variable **STREET_TYPE** begins in column 21 for a length of 10 and spaces represent a missing value.

Variable names must contain 1–16 characters with the first character being alphabetic. The underscore character is permitted as part of the name. All commands and variable names are case-insensitive. Consequently, "**STREET_NAME**" is equivalent to "**Street_Name**". Variables may be named the same on any file if so desired.

Match Specifications

The match specification file contains a number of MATCH commands. In general, the order of the commands should be the same as the order that they are presented in the following discussion.

The MATCH commands specify the variables that will participate in a match. Up to 40 MATCH commands may be specified for a pass.

It is a good idea to include all variables in common on both files as MATCH variables. Often, people feel that variables that are not very reliable should be omitted. It is best to include even

variables with a low m-probability so there is a smaller penalty for mismatches.

You will notice that a matching run on fewer variables will have better match rates than those with more variables included. This is because decreasing the number of variables that participate in a match also decreases the discriminating ability to differentiate the correct matches from the incorrect matches. For example, matching a pair of files on the variable sex alone will match almost all the records, but you would have no confidence that the matches pertain to the same individuals on both files.

The MATCH statements form the heart of a matching run. The format of the MATCH statements may seem complex, but basically they specify the type of comparisons to be made (character, numeric, and so on), the fields involved in the comparisons, and the m and u probabilities.

MATCH statements are formatted according to the following:

For matching single variables:

MATCH

<comparison-type>

<match key field>

<reference file variable name>

<m-probability>

<u-probability>

[<additional-parameters>]

[<mode>]

where:

<comparison-type> ::=

CHAR: Character

CNT_DIFF: Count number of chars difference

D_INT: Left/Right Intervals (DIME/TIGER)

DATE6: Date in the form of yymmdd

DATE8: Date in the form of yyyyymmdd

DELTA_PER: Delta Percent

DISTANCE: Geometric distance

INTERVAL_NOPAR: Interval

INTERVAL_PARITY: Odd/Even Interval

LR_CHAR: Left/Right character string comparison

LR_UNCERT: Left/Right uncertainty string comparison

NUMERIC: Numeric

PREFIX: Prefix character

PRORATED: Prorated

TIME: Time

UNCERT: Uncertainty character comparisons

<additional-parameters>

a numeric value or value required for types **UNCERT**, **LR_UNCERT**, **PRORATED**, **TIME**, and **DELTA_PER**.

<mode>: required for types **INTERVAL_NO**, **INTERVAL_PARITY**, and **D_INT**:

ZERO_VALID: Zero is valid and may be part of interval

ZERO_NULL: Zero is null and never part of the interval required for types **LR_CHAR** and **LR_UNCERT**:

EITHER: Match on either left or right side

BASED_PREV: Match on side based on results of previous left/right interval compare

Please note that not all comparison types are applicable for ArcView geocoding applications. They are included here for reference only.

Specifying the m and u probabilities

The MATCH commands require the m and u probabilities. These are defined as:

m probability: The probability that the field agrees given the record pair is a match. This is essentially one minus the error rate of the field in matched records.

u probability: The probability that the field agrees given the record pair is unmatched.

Since, given all possible record pairs, the number of unmatched pairs is much greater than the number of matched pairs, the u probabilities can be restated as:

u probability: The probability that the field agrees at random.

Initially, the user should estimate a u probability. The following can serve as a guideline:

Sex: u probability = 0.5

Age: u probability = 0.02

Similarly, the user should provide an initial estimate of the m probabilities. Values of 0.9 to 0.99 are typical. The closer this value is to one, the more critical a disagreement on the field becomes. This can be used to make fields that are important have a high penalty for mismatching.

The weight for a field is calculated as follows:

If the fields agree then the weight is the log to the base 2 of the ratio of the m and u probabilities.

If the fields disagree, then the weight is the log to the base 2 of the ratio of 1-m and 1-u.

Fields that agree receive positive weights, and fields that disagree receive negative weights. The composite weight for the record pair being examined is the sum of all the individual field weights.

Examples:

```
MATCH LR_UNCERT ZN ZipLeft ZipRight 0.9 0.01 700.0 EITHER
```

```
MATCH UNCERT SN StreetName 0.9 0.01 700.0
```

```
MATCH CHAR PD PreDir 0.8 0.1
```

```
MATCH CHAR PT PreType 0.7 0.1
```

```
MATCH CHAR ST SuffixType 0.85 0.1
```

```
MATCH CHAR ST SuffixDir 0.85 0.1
```

```
MATCH D_INT HN FromLeft ToLeft FromRight ToRight 0.999  
0.05 ZERO_VALID
```

Notice the m and u probabilities and the additional parameter where required.

Character comparisons (CHAR)

Character (CHAR). This is a character-by-character comparison. If one field is shorter than the other, the shorter field will be padded with trailing blanks internally to match the length of the longer field. Any mismatched character causes the disagreement weight to be assigned.

Example:

```
MATCH CHAR STREET_NAME STREET 0.9 0.02
```

The variable **STREET_NAME** on file **A** is compared to the variable **STREET** on file **B**.

Counting errors in fields (CNT_DIFF)

Count Difference (CNT_DIFF) is used to count keying errors in numeric fields. For example, suppose a birth date appears in both files as:

19670301

19670801

There is a strong possibility that these represent the same birthdate with a keying error on the month (**03** versus **08**). The **CNT_DIFF** comparison type can be used to take this type of error into consideration. **CNT_DIFF** would generally be used on numeric fields, such as birth dates, telephone numbers, file or record numbers, Social Security Numbers, and so on. For character fields, such as name, street name, and so on, it is better to use **UNCERT**.

The **CNT_DIFF** variable type requires an additional parameter to indicate how many keying errors will be tolerated. If **1** is specified, then no errors would result in the agreement weight being assigned, one error would result in assigning the agreement weight - 1/2 (agreement weight + disagreement weight), and two or more errors would result in the disagreement weight (remember that the disagreement weight is a negative number). Thus, one error would yield a partial weight. If **2** is specified, then the errors are divided into thirds. One error would receive the agreement weight minus 1/3 the weight range from agreement to disagreement, two errors would receive the agreement weight minus 2/3 the weight range, and so on. Thus, the weights are prorated according to the seriousness of the disagreement.

Consider the following example of the **CNT_DIFF** comparison type:

```
MATCH CNT_DIFF Phone Phone 0.9 0.0001 1
```

Here we will tolerate one keying error in the phone number.

Date comparison types

Use **Date Comparisons (DATE6 and DATE8)** when you need to tolerate differences in dates, taking into account the various number of days in a month, leap years, and so on. Two types of dates may be processed. Six character dates (DATE6) have the **19** or **20** of the year omitted and are in the format **yyymmdd**. Eight character dates (DATE8) have the entire year specified: **yyyymmdd**. The **y** specifies character positions for year, **m** specifies character positions for month, and **d** specifies character positions for day.

For example, 19820103 is January 3, 1982 in the DATE8 format, and 820103 is the same date in the DATE6 format.

The date comparison types require one or two extra parameters. If one parameter is specified, then this is the number of days difference that can be tolerated. For example:

```
MATCH DATE8 BirthDate Birth 0.9 0.01 1
```

If the birthdate agrees on both files, then the agreement weight is assigned. If they differ by one day, then the weight is the agreement weight minus 1/2 of the weight range from agreement to disagreement. Two or more days' difference results in a disagreement weight. Similarly, if the value is 2, then a one-day difference reduces the agreement weight by 1/3 of the weight range, two days reduces the agreement weight by 2/3, and so on.

Two parameters may be coded if different tolerances were desired in each direction. For example, in matching highway crashes to hospital admissions, an admission can occur before the accident date and be related to the accident:

```
MATCH DATE8 CrashDate AdmitDate 0.9 0.01 1 0
```

The first additional parameter is the tolerance if the file B value is greater than the file A value, and the second parameter is the tolerance if the B value is less than the A value. Thus, in this

example, a difference of one day is tolerated if the B value is greater (admit date greater than crash date), and a difference of zero is tolerated if the reverse is true (admit date earlier than the crash date).

Delta percentage comparisons

You can use DELTA_PER to compare fields, such as age, where the difference should be measured in percent. For example, a one-year difference in an 85-year-old is much less serious than a one-year difference in a 3-year-old. The additional parameter specifies the maximum percentage difference tolerated. The base of the percentage is the minimum value of A or B:

$$(| \text{Value B} - \text{Value A} | / \min(\text{A}, \text{B})) * 100$$

Example:

MATCH DELTA_PER AGE 0.9 0.05 10

This example tolerates a 10 percent difference in the AGE variables.

The weight assigned to the match is prorated on the difference between the values: a one-percent difference subtracts 1/11 of the weight range (the difference between the agreement and disagreement weight) from the agreement weight. A 10 percent difference subtracts 10/11 of the difference in the weight range. Thus, a 10 percent error is barely tolerated, but an 11 percent error is not.

Two parameters are permitted if different tolerances are desired for different directions of difference. For example, MATCH DELTA_PER AGE 0.9 0.05 10 5 tolerates a 10 percent error if the B value is greater than the A value, and a five percent error if the reverse is true.

Distance comparisons (DISTANCE)

The distance comparison type is used to compute the Pythagorean distance between two points and prorate the weight on the basis of the distance between the points. This comparison type requires four field names and a tolerance parameter:

MATCH DISTANCE <A-X> <A-Y> <B-X> <B-Y> <max-distance>

<A-X> is the field name that contains the x coordinate on file A. **A-Y** is the y coordinate of file A, **B-X** is the x coordinate on file B, and **B-Y** is the y coordinate on file B. **max-distance** is the maximum distance to be tolerated. The distance is in the units of the coordinates. For example, if the coordinates are in thousandths of a degree, then a max distance of **100** would tolerate a distance of **0.1 degrees**. If the distance between the points is zero, then the agreement weight is assigned. If the distance is 0.05 degrees, then the midpoint between the agreement and disagreement weight is assigned. If the distance is 0.1 degree or greater, then the disagreement weight is assigned.

This comparison type allows use of coordinates to select the closest record. For example, if you have ZIP Code centroid coordinates, and you have a choice of matching to several ZIP Codes, you would want to choose the closest ZIP Codes first.

Frequencies are not computed for distance variables.

The coordinates must be positive or negative integers. Decimal places are not permitted.

Interval comparisons

Interval comparisons (INTERVAL_NOPAR) are used to compare a single number on file A to an interval (range of numbers) on file B. Interval comparison types are primarily used for geocoding applications. In these cases, file B is the reference file. The single number must be within the interval (including the endpoints) to be considered a match. Otherwise, it is a disagreement. Interval

comparisons require a mode parameter. The mode indicates whether zero or blanks should be treated as any other number (**ZERO_VALID**), or if zero or blank fields should be considered null or missing values (**ZERO_NULL**). This type of comparison requires three variable names: the number on file **A**, the begin range of the interval on file **B**, and the end range of the interval on file **B**. Since interval comparisons are used primarily for geocoding, the number on file **A** will match the interval on file **B** even if the begin range is higher than the end range. For example, 153 will match 200–100 and will also match 100–200.

Interval comparison with parity

The Odd/Even Interval (**INTERVAL_PARITY**) type is identical to the **INTERVAL_NO** type comparison, except that the number must agree in parity with the low range of the interval. A single number on file **A** is compared to an interval on file **B**. If the number on file **A** is odd, then the begin range number on file **B** must also be odd to be considered a match. Similarly, if the number on **A** is even, then the begin range on **B** must be even. This type of comparison is used primarily for geocoding applications in comparing a house number on file **A** to a range of addresses on file **B**. Reference files, such as the ZIP+4 files, have a single odd or even interval. Odd/Even comparisons require a mode parameter. The mode indicates whether zero or blanks should be treated as any other number (**ZERO_VALID**), or if zero or blank fields should be considered null or missing values (**ZERO_NULL**). Zero is considered to be an even number. This type of comparison requires three variable names: the number on file **A**, the begin range of the interval on file **B**, and the end range of the interval on file **B**. The number on file **A** will match the interval on file **B** even if the begin range is higher than the end range. For example, 153 will match 199–101 and will also match 101–199.

Example:

```
MATCH INTERVAL_PARITY HOUSE FROM TO 0.999 0.01  
ZERO_VALID
```

The field **HOUSE** on file **A** is compared to the interval **FROM** and **TO** on file **B**. Zero is considered as any other number. Note that the **m** probability is high to force a mismatch if the house number disagrees.

Left/Right interval comparisons (**D_INT**)

This type of comparison is primarily used to compare house numbers in Census Bureau TIGER files, the GDT DynaMap files, or the U.S. Post Office ZIP+4 files. A single house number is compared to two intervals. One interval represents the left side of the street and the other represents the right side of the street. In order for a number to match to an interval, both the parity (odd/even) and the range must agree. This comparison causes a special flag to be set to indicate whether the left or the right interval is matched. The **MOVELR** statement of the extract program can be used to move the proper geocodes to the output record. A separate section of this manual discusses geocoding applications in detail. The begin number of the intervals may be higher than the end number and still match. This is because the TIGER files may have a high address in the **FROM** position and a low address in the **TO** position, depending on the orientation of the segment.

Example:

```
MATCH D_INT HOUSE FROM_LEFT TO_LEFT FROM_RT  
TO_RT 0.999 .01 ZERO_VALID
```

The field **HOUSE** is compared to the left interval (**FROM_LEFT**, **TO_LEFT**) and the right interval (**FROM_RT**, **TO_RT**). The left or right flag is set depending on the results.

Left/Right character comparisons (LR_CHAR)

This type of comparison is used in conjunction with geocoding applications for comparing place or ZIP Code information. Census Bureau TIGER files and other geographic reference files contain a left ZIP Code and a right ZIP Code, a left city code and a right city code, and so on. The left code applies if there was a match to the left address range interval, and the right code applies if there was a match to the right address range. A single field on user data file must be matched to the two fields on file **B** on a character-by-character basis. A single field is specified on the **A** file and two fields are specified on the **B** file. There are two modes of comparisons. The first mode **EITHER** requires that the contents of the file **A** field match either of the file **B** fields specified (or both) to receive the full agreement weight. Since persons living on a boundary segment often report the neighboring code, an either/or match on the code is preferable to rejecting the record. The second mode, **BASED_PREV**, uses the result of a previously defined **D_INT** (double interval) comparison result to decide which field to compare. If the address range specified in a **D_INT** comparison matched to the left interval, then the file **A** field must match to the first file **B** field specified to receive the agreement weight. If the comparison matched to the right interval, then the file **A** field must match the second field specified. Thus, if the left interval matched, the left code must agree, and if the right interval matched, the right code must agree. If neither the left or right interval agrees, the missing weight for the field is assigned.

Example:

```
MATCH LR_CHAR ZIP_CODE LEFT_ZIP RIGHT_ZIP 0.9 .01  
EITHER
```

The **ZIP_CODE** field on File **A** may match to either the **LEFT_ZIP** field on file **B** or the **RIGHT_ZIP** field on file **B** (or it may match to both).

Example:

```
MATCH D_INT HOUSE FROM_LEFT TO_LEFT FROM_RT  
TO_RT 0.999 .01 ZERO_VALID
```

```
MATCH LR_CHAR ZIP_CODE LEFT_ZIP RIGHT_ZIP 0.9 .01  
BASED_PREV
```

This is different from the previous example because the mode is **BASED_PREV** instead of **EITHER**. The **D_INT** compare on **HOUSE** will set the **L/R** flag, depending on whether the left or right interval matched. The **LR_CHAR** compare uses the result of the match to decide which field to compare. If the **L** flag is set, the **ZIP_CODE** field on File **A** must match the **LEFT_ZIP** field on file **B**. If the **R** flag is set, the **ZIP_CODE** must match to **RIGHT_ZIP** field on file **B**. If neither flag is set, the missing weight is assigned to the comparison. It is important that the type **D_INT** compare precedes the type **LR_CHAR** compare with mode **BASED_PREV**.

Left/Right uncertainty character comparisons (LR_UNCERT)

This type of comparison is used in conjunction with geocoding applications for comparing place or ZIP Code information. Census Bureau TIGER files and other geographic reference files contain a left ZIP Code and a right ZIP Code, a left city code and a right city code, and so on. This operates identically to the type **LR_CHAR** comparison, except that the type **UNCERT** uncertainty character algorithm is used. The threshold parameter is required before the mode, which must be **EITHER** or **BASED_PREV**.

Example:

```
MATCH LR_UNCERT CITY_NAME LEFT_CITY RIGHT_CITY  
0.9 .01 700.0 EITHER
```

The **CITY_NAME** field on file **A** may match to either the **LEFT_CITY** field on file **B** or the **RIGHT_CITY** field on file **B** (or

it may match to both), using the uncertainty character comparison with a minimum threshold value of 700.

Numeric comparisons (NUMERIC)

This is an algebraic numeric comparison. Leading spaces are converted internally to zeros and the numbers are compared.

Example:

```
MATCH NUMERIC AGE AGE 0.9 0.05
```

Prefix comparisons for truncated fields (PREFIX)

The **PREFIX** comparison type is useful for performing character matching on fields that may be truncated. For example, a last name of **ABECROMBY** could be truncated to **ABECROM** because of lack of space. The **PREFIX** comparison type will consider these two representations to be an equal match. The string length to be compared is the shorter of the two strings. No characters are compared after the length of the shorter string. This is different from **CHAR**, where these two names would not match. The length is computed by ignoring trailing blanks (spaces). Embedded blanks are not ignored.

Prorated comparisons (PRORATED)

This allows numeric fields to disagree by a specified absolute amount as specified by the additional parameter. For example, if the parameter is 15, then if the absolute value of the difference in the field values is greater than 15, the disagreement weight would be assigned to the comparison. If the difference is zero, then the full agreement weight would be assigned. Any difference between 0 and 15 would receive a weight proportionally equal to the difference. Thus, small differences would be slightly less than the full agreement weight, while large differences would be slightly greater than the full disagreement weight. A difference of 8 would

receive a weight exactly between the agreement and disagreement weight.

Example:

```
MATCH PRORATED DAY DAY 0.90 0.1 5
```

The field **DAY** on file **A** is compared to the field **DAY** on file **B**. The numeric values may differ by as much as 5. Thus, the weights are prorated from agreement to disagreement from differences between 0 and 5. (This is equivalent to saying **DAY** + or - 5).

Two additional parameters may be specified if it matters whether the difference is positive or negative. The first parameter is the tolerance if the value on file **B** is greater than the value on file **A**. The second parameter is the tolerance if the value on **A** is greater than the value on **B**. For example:

```
MATCH PRORATED Aday Bday 0.9 0.1 5 7
```

The **B** value may exceed the **A** value by 5 days, but the **A** value may exceed the **B** value by 7 days.

Time comparisons (TIME)

This compares times in the form of hours and minutes (or only hours). The time must be in 24-hour format. 0 is midnight and 2359 is 11:59 PM. An additional parameter is required to indicate the maximum acceptable time difference. For example, 60 would tolerate a difference of 60 minutes. Again, the weights are proportionally divided between the full agreement weight and the full disagreement weight. Times may cross midnight. The difference is always the shortest way around the clock. The difference between 2300 and 0100 is 120 minutes.

Example:

```
MATCH TIME ACCID_TIME ACCID 0.9 0.2 60
```

The field **ACCID_TIME** on file **A** is compared to the field **ACCID** on file **B**. A difference of 60 minutes is allowed. Therefore, the comparison is equivalent to saying time + or -60 minutes.

A second parameter may be added if unequal tolerances are desired. The parameter specifies the tolerance if the value on the **B** file is greater than the value on the **A** file. The second tolerance is for the reverse situation.

For example, suppose file **A** contains a highway crash time and file **B** contains the arrival time of an ambulance:

```
MATCH TIME CrashTime AmbulanceTime 0.9 0.2 60 5
```

The ambulance time can be 60 minutes later than the crash time, but the crash time can only be five minutes later than the ambulance time. This allows for minor errors in recording the times.

Character uncertainty comparisons (UNCERT)

This uses an information-theoretic character comparison algorithm. It tolerates phonetic errors, transpositions, random insertion, deletion, and replacement of characters. The weight assigned is based on the difference between the two strings being compared. An additional parameter is required if type **UNCERT** fields are specified. This is the cutoff threshold and is a number between 0 and 900. The following guidelines apply:

900: The two strings are identical

850: The two strings are so close they can be safely considered to be the same.

800: The two strings are probably the same.

750: The two strings are probably different.

700: The two strings are almost certainly different.

700 is a good value for this parameter. This causes the full disagreement weight to be assigned if the score is 700. Otherwise, the weight assigned is proportioned linearly between the agreement and disagreement weights.

The score assigned is a function of the string length (since longer words can tolerate more errors and still be recognizable than shorter words can); the number of transpositions; and the number of unassigned insertions, deletions, or replacements of characters.

Example:

```
MATCH UNCERT LAST_NAME LAST 0.90 0.01 700.0
```

In this example, the field **LAST_NAME** on file **A** is compared to the field **LAST** on file **B** using the uncertainty comparison algorithm with a threshold of 700.

VARTYPE

Sometimes fields require special treatment. The **VARTYPE** commands are used to indicate this. There may be more than one **VARTYPE** statement for the same variable. Its format is:

VARTYPE

<file A variable name>

<action>

where

<action> ::=

CRITICAL

CLERICAL

NOUPDATE

NOFREQ

CRITICAL: a critical field is one where a disagreement on the field causes the record pair to automatically be considered a nonmatch. For instance, you wouldn't tolerate an error on birth date, so this field can be made critical.

CLERICAL: a clerical field is one where a disagreement on the field causes the record pair to automatically be considered a clerical review case regardless of the weight. For example, in motor vehicle accident matching, if the county code does not match, the records should be clerically verified.

NOUPDATE: it is sometimes the case that the **m** probabilities are chosen to artificially force a match. For example, giving a high **m** probability penalizes disagreements severely. For example, a motor vehicle accident flag might be virtually critical to the match—above the level indicated by its real probability. Consequently, the **MPROB** program should not change the **m** probability for these cases.

NOFREQ: some fields have unique values (such as Social Security Number), and it is a waste of time to conduct a frequency analysis of these fields. **NOFREQ** indicates that no frequency analysis should be performed.

CONCAT: it is sometimes desirable to concatenate variables to form one frequency count. For example, diagnostic codes may be highly correlated to age and sex. It is not likely that a 20 year-old male will develop breast cancer. As many as four variables may be concatenated:

VARTPYE Diag CONCAT Age Sex

The values of **Age** and **Sex** will be concatenated to the value for **Diag**. When any one of these variables participates in a match, the values specified are concatenated and looked up in the frequency table. Thus, **Diag, Age, Sex** form one value in the table.

Consider these other examples of **VARTYPE**:

Example:

VARTYPE FIRST_NAME CLERICAL

First name is a clerical field. All errors on first name should be forced into clerical review.

Example:

VARTYPE SSN NOUPDATE

VARTYPE SSN NOFREQ

Neither frequency analysis nor updating is desired for the Social Security Number.

Geocoding

Geocoding involves matching a data file to a geographic reference file. This differs from individual matching since each data record is independent, and a single reference file record can participate in multiple matches. Except for a few exceptions, the same statements used for a geocoding case are used for any other case.

It is often necessary to match a data file record to a reference file, such as the U.S. Census Bureau's TIGER File. These files have two sets of address ranges: one representing the left side of the street segment and the other representing the right side of the street segment. Odd house numbers will generally be on one side of the street and even numbers on the other side. A single house number on the data file will match to only one of these intervals. The matcher handles this case as follows:

Define a type **D_INT** field in the match specifications field. This requires a house number field on file **A** and four house number fields on file **B**. These are the from-left number, the to-left number, the from-right number, and the to-right number. Only one type **D_INT** field is permitted per matching pass.

When the match is executed, the matcher sets the **L/R** flag to indicate whether the match was to the left interval, the right interval, or neither.

If a city or ZIP Code must be matched to either a left or right city or ZIP Code, use the type **LR_CHAR** or type **LR_UNCERT** comparisons. This has two modes. The **EITHER** mode requires the code on file **A** to match to either the left or right field. This is useful because people living on boundaries may mistakenly code the incorrect city or ZIP. The **BASED_PREV** mode uses the result of a prior **D_INT** comparison to decide to which field to match. If the **L** flag is set, the left code must match; if the **R** flag is set, the right code must match; and if neither flag is set, the missing weight is assigned.

It is advisable to give the type **D_INT** field a high **m** probability in the match specification, such as 0.999, to force the record to be unmatched if the house number does not agree. Also, since many house numbers are possible, a frequency analysis of house numbers is probably not useful, so the **VARTYPE NOFREQ** parameter should be used. Example cases for geocoding are supplied with the matcher.

Finally, you must be sure to standardize the addresses before attempting to match. This should be performed with an address standardization program. The **AUTOSTAN** product can be used for this purpose. An address standardizer creates fixed fields for each component of the address (house number, directions, street name, street type, and so on) and standardizes spellings of common abbreviations (**ST.**, **PL.**, and so on).

Record linkage concepts

This section presents a tutorial on record linkage concepts. It is written for a general audience.

The discussions below apply to the individual matching problem, but the same arguments also apply to unduplication and geocoding.

Objective of record linkage

Individual record linkage involves two files: **File A** and **File B**. It generally doesn't matter which file is which; however, if file **A** is generally the master or larger file, then analysis is somewhat easier. The objective of the record linkage (or matching) process is to identify all records in one file that correspond to the same individual, event, household, street address, and so on in the second file. When a single file is unduplicated, then all records in that file pertaining to a single individual, address, and so on, are grouped together. Individual matching is different from geocoding because once an individual is matched, those records cannot match to anyone else, since they pertain to a single individual. Geocoding involves a reference file, which must be file **B**, that supplies, for example, a range of addresses for a postal code. More than one person on file **A** can match to a single record on the reference file. If a single reference file record stated that all households from **101 to 9899 MAIN ST** were in ZIP Code **98765**, then input records for **101 MAIN ST**, **1235 MAIN ST**, and so on, would all match to this record.

Each file consists of fixed **fields** that contain the information to be matched. The fields must be in fixed locations and have a uniform size. Scientific methods of record linkage are required since all fields contain errors or missing values, and it is best to find the matches with a reasonable statistical assurance.

Obviously, one or more fields in file **A** must have equivalent fields in file **B**. For example, to match surname and age, both files **A** and **B** must contain fields containing this information. The location

and length of a field in file **A** may be different from its equivalent field in file **B**.

If you were to conduct a record linkage experiment, you could create a set of all possible record pairs. The first pair would be record one from file **A** with record 1 from file **B**. The next pair would be record one from file **A** and record 2 from file **B**, until **$n \times m$** pairs were formed, where **n** is the number of records on file **A** and **m** is the number of records on file **B**.

The objective of the record linkage process is to classify each pair as belonging to one of two sets: the set of matched record pairs, **M**, and the set of unmatched record pairs, **U**. For example, if you were to inspect the pair created from record **123** in file **A** with record **217** in file **B**, you must be able to say it is not a match and belongs in set **U**, or it is a match and belongs in set **M**.

Obviously, there are many more unmatched pairs than matched pairs. To illustrate this, consider two files with 1,000 records each. There are 1,000,000 possible record pairs, but only 1,000 possible matches if there are no duplicates in the files. Thus, set **M** will contain at most 1,000 pairs, and set **U** will contain the remaining 999,000 pairs.

Feasibility of record linkage

In order for a record linkage application to be feasible, it should be possible for a human to examine the match fields for any record in file **A** and the equivalent fields for any record in file **B**, and declare with reasonable certainty that the record pair examined is a match or a nonmatch.

For example, if the only field in common between two files was **sex**, no one would say that if the sex agreed, then the pair represented the same individual. However, if both files contained a field such as Social Security Number, then you could claim that if there was a match, it represented the same individual.

A rule of thumb, to determine if a record linkage application is feasible, is to multiply the number of values in each field, and compare this product with the number of records in both files. If the product is much greater than the number of records, the application is probably feasible.

For example, if the fields **sex**, **age**, and **middle initial** were the only fields that could serve as matching fields, then the following calculation could be made: **sex** has two possible values, **age** has one hundred, and **middle initial** has twenty-six. When you multiply the possible values in this field, you get 5,200 ($2 \times 100 \times 26$). Since there are only 5,200 possible values for the fields, only very small datasets can be matched with any confidence. The probability that more than one record is an exact duplicate and does not represent the same individual is high with a file size of 5,200. The actual probabilities would depend on the distribution of the values in the fields.

Blocking

For any reasonably sized file, it is unreasonable to compare all record pairs since the number of possible pairs is the product of the number of records in each file. Even a case with two small files of 1,000 records each has 1,000,000 possible pairs to examine. Of this million, a maximum of 1,000 will be matches. The other 999,000 are unmatched pairs. If there were a way to look at pairs of records having a high probability of being matches and ignoring all pairs with low probabilities, then it would become computationally feasible to conduct the linkage with large files.

Fortunately, the concept of **blocking** provides a method of limiting the number of pairs being examined. If one were to partition both files into mutually exclusive and exhaustive subsets and only search for matches within a subset, then the process of linkage would become manageable.

To understand the concept of blocking, consider a field such as **age**. If there are 100 possible ages, then this variable partitions a file into 100 subsets. The first subset is all people with an age of zero, the next is those with an age of 1, and so on. These subsets are called **blocks** (or **pockets** in some systems). Suppose, for the sake of example, that the age values were uniformly distributed. In this case, out of the 1,000-record file, there would be 10 records for people of age 0 on each file, 10 records for people of age 1, and so on.

The pairs of records to be compared are taken from records in the same block. The first block would consist of all persons of age 0 in files **A** and **B**. This would be 10×10 or 100 record pairs. The second block would consist of all persons in files **A** and **B** with an age of 1. When the process is complete, you would have compared 100 (blocks) \times 100 (pairs in a block) = 10,000 pairs, rather than the 1,000,000 record pairs required without blocking.

Blocking ensures that all records that have the same value in the blocking variable are compared. One consequence of this is that records that don't match on the blocking variables will automatically be classified as nonmatched. For example, if our blocking variable were age, and age was in error in one of the files, then the records involved are considered to be unmatched. To get around this problem, **multiple passes** are used.

Suppose a match is run where age is the blocking variable. Any records that do not match can be rematched using another blocking scheme, for example, postal code of residence. If a record did not match on age in pass 1, then it still has an opportunity to match on postal code in pass 2. It is only those cases that have errors in both the age and postal code fields that will not be matched. If this is a major problem, then a third pass can be run with different blocking variables. Errors on all three blocking variables are unlikely.

Selecting blocking strategies

It should be obvious from the example above that smaller blocks are many times more efficient than larger blocks. It is much better to use restrictive blocking schemes, especially in the first pass. Since most of the records will match on the first pass, a second pass match has fewer records to process and can be less restrictive.

Using a variable such as age alone is not a good blocking strategy, since age is generally not uniformly distributed (some ages may be much more prevalent in the files than others), and partitioning a large file into 100 categories still leaves many records in each block.

You can use more than one variable as a blocking variable in a pass. For example, if **age** and **sex** are both blocking variables and sex is coded as M or F, then the first block would be zero-year-old females, the second would be zero-year-old males, then one-year-old females, one-year-old males, and so on. This would partition the files into 200 subsets (100 ages x 2 sexes).

Consider a pair of files containing the following fields:

Surname

Middle Initial

Given Name

Sex

Birthdate (year, month, and day)

The first pass can be blocked on **surname**, **sex**, and **birthyear**. The second pass can be blocked on **birthmonth**, **birthday**, the initial character of the **Given Name**, and **Middle Initial**.

The matching process would group all people with the same surname and sex born in the same year. Any error on one or more of these variables would be picked up in pass 2, which would

include all people that have the same birth month and day with the same first initial characters of their given name and middle initial.

AUTOMATCH will automatically control multiple pass matching schemes like this. The unmatched records from pass 1 are called residuals. These residuals become the input files for the pass 2 match, and so on for all passes. However, in general, two passes are sufficient to match almost all cases.

The blocks should be as small as possible. Ten to 20 records per file is a good block size. Blocks should never exceed 100 records per file, or efficiency will be quite poor. The PC version of the program has a limit of 65,536 pairs in a block. The largest square block permitted would have 256 records in file **A** compared to 256 records in file **B**. The largest nonsquare block would have 5,000 records in file **A** and 12 records in file **B**, or vice versa. Any combination of sizes that doesn't exceed 65,536 pairs or 5,000 records on any single file is permissible for block sizes.

If the maximum block size is exceeded, then all of the records in the block are skipped. They become residuals to be processed in pass 2. This is called a block **overflow**.

The variables that are the best blocking variables are those with the most number of values possible and the highest reliability. For example, **sex** alone is a poor choice, since it only divides the file into two subsets. Similarly, fields subject to a great probability of error should be avoided. For example, **apartment number** is generally misreported or omitted, and hence would not make a good blocking variable. In mathematical terms, the fields with the highest **weights** make the best blocking variables.

Weights

The information contained in the variables to be matched helps the matcher, or a human, determine which record pairs are matches and which are nonmatches. Each field provides some information. Taken together, all the fields should determine, with little equivocation, the status of the pair being examined.

Some fields provide more information more reliably than others. For example, it would be absurd to sort both files on the **sex** variable and assert that if the sex agrees, the record pair represents the same individual. However, it would make sense to sort both files on Social Security Number and assert that if this number agrees, then the record pair represents the same individual. This section discusses how the **discriminating power** of each variable can be measured.

Each field has two probabilities associated with it. These are called the **m** and **u** probabilities. The **m** probability is the probability that a field agrees, given that the record pair being examined is a matched pair. This is effectively one minus the error rate of the field. For example, in a sample of matched records, if sex disagrees 10 percent of the time due to a transcription error or because it is misreported, then the **m** probability for this variable would be 0.9 (1 - 0.1).

The more reliable a field is, the greater the **m** probability will be.

The **u** probability is the probability that a field agrees given that the record pair being examined is an unmatched pair. Since there are so many more unmatched pairs possible than matched pairs, this probability is effectively the probability that the field agrees at random. For example, the probability that the sex variable

agrees at random is about 0.5. Given a uniform distribution, there are four possibilities:

<u>File A</u>	<u>File B</u>
M	F
M	M
F	M
F	F

The sex agrees in two of the four combinations (thus, 0.5 **u** probability).

The user must provide a guess for the **m** and **u** probabilities for each field.

The weight for a field is computed as the logarithm to the base two of the ratio of **m** and **u**. To see how this translates into actual values, you can examine the example of the sex and Social Security Number variables:

Assume that sex has a 10 percent error rate and Social Security Number has a 40 percent error rate.

The **m** probability for sex is 0.9. The **u** probability is 0.5 (from the above table). Thus, the weight for sex is $\log_2(m/u) = \ln(m/u)/\ln(2) = \ln(0.9/0.5)/\ln(2) = 0.85$.

Conservatively, assume that the probability of chance agreement of Social Security Numbers is one in 10 million. Given **m** as 0.6 (40 percent error rate in matched pairs), then the weight for Social Security is $\ln(0.6/0.0000001) = 22.51$.

Thus, the weight for a match on the sex variable is 0.85 and a match on the Social Security Number is worth 22.51. The weights have captured what we know intuitively about the variables.

Composite weights

For each record pair compared, a composite weight is computed and stored. The composite weight is the sum of the individual weights for all field comparisons.

If a field agrees in the pair being compared, the agreement weight, as computed above, is used. If a field disagrees in the pair being compared, the disagreement weight is computed as: $\log_2 [(1-m)/(1-u)]$. This results in field disagreements receiving negative weights. Thus, agreements add to the composite weight and disagreements subtract from the composite weight. The higher the score, the greater the agreement.

The matcher program produces a histogram of the distribution of the composite weights. This histogram has many values at highly negative weights because most cases are unmatched pairs; however, the matcher does not spend much time comparing records that disagree, and thus, negative weights are not often shown. There is another “mode” at highly positive weights for the matched cases. The cutoff value for the match run can be set by inspecting this histogram. The clerical review cutoff should be the weight where the “bump” in the histogram reaches near the axis, and the other cutoff weight should be where the unmatched cases start to dominate. Experimentation and examination of the results are the best guide for setting the cutoffs.

Estimating probabilities

You can estimate the m and u probabilities even if you have no idea of the appropriate values. One value that is always a good guess for the m probabilities is 0.9. The higher the m probability, the greater the disagreement weight will be. Therefore, if a field is important, the m probability can be a higher value. If the m probability is high, it means that a disagreement of that field is a rare event in a matched pair, and consequently the penalty for a

nonmatch is high. The weights computed from the probabilities are printed by the matcher, so the results can be inspected.

Give the fields that are most important and reliable high m probabilities, and give the fields that are often in error or incomplete lower m probabilities. The m probability must always be greater than the u probability, and it must never be zero or one.

The u probability can be guessed. A good starting guess is to make the u probability $1/n$ values, where n values is the number of unique values for the field. For example, sex has two values, so a good guess for the u probability is $1/2 = 0.5$.

It is important to know that even exact matching is subject to the same statistical laws as probabilistic matching, since it is possible to have two records match identically and not represent the same individual.

Endnote

With permission from MatchWare Technologies, Inc., text for this appendix was adapted from the AUTOMATCH User’s Manual (version 2.9) by ESRI, 380 New York St., CA 92373, U.S.A.

Standardization process syntax

B

IN THIS APPENDIX

- **Introduction**
- **Input file format specifications**
- **The match key**
- **The classification table**
- **The pattern rules**
- **Unconditional patterns**
- **Conditional patterns**
- **Actions**
- **Summary**

This appendix contains the documentation for preparing an address standardization process. A standardization process includes the specifications of a match key dictionary, a classification table, and a pattern rule and action file.

Introduction

Individual names, postal addresses, and company names often appear in computer files in a free format. For example, a single field named Address may contain 123 Main St, Apartment 103. A name field could contain Mr. John Paul Jones.

The information in these fields must be standardized if conformance to postal service standards is required or if it is necessary to recognize or process the information. Processes requiring standardization include address matching, individual matching, mailing list duplication, and so on.

The standardization process involves the creation of a **match key**. This match key consists of individual fields for each element of the name or address. The standardizer program must scan the free format information, recognize each element, and move it to the appropriate field in the match key. In addition, keywords have standardized abbreviations.

The output of the standardizer is the newly created match key, with all of its individual fields followed by the original input record. These concepts are best illustrated by several examples.

STANEDIT is the Windows version of the Interactive Standardizer, and it is used for pattern rules syntax checking and debugging. The discussion in this appendix focuses on STANEDIT, as it allows for testing and debugging pattern rules, and also prompts for interactive input.

Postal addresses

Fields containing postal address information are the most complex of all those mentioned. Addresses come in a variety of forms, and recognizing the elements is not as easy as counting words or looking for particular key words. For introductory purposes, consider this simplified match key:

House number

Direction

Street Name

Street Type

Apartment

Now, consider several sample addresses:

123 Main St

101-E E E Street

1203 St Charles St #4

345 East West Highway Apt A34

97 North West St

It is not always apparent to what a keyword refers. For example, a keyword such as **E** doesn't necessarily refer to the direction **EAST**, since this can also be used to refer to an apartment number or a street name. Street names can also contain directions, and the proper interpretation may be dependent on the area in which these cases are found. For example, **EAST-WEST** is the street name for **EAST WEST HWY**; however, for **NORTH WEST ST**, **NORTH** is the direction and **WEST** is the name of the street.

Files used by STANEDIT

STANEDIT uses the following files:

<output-file>: the output from STANEDIT contains the patterns that are matched and match keys generated from the standardization process.

<command-file>: the command file is an ASCII text file that the user must prepare with a standard text editor, such as Notepad. It contains the description of the processes desired, the formats of the input files, and so on. This file is required for every run of STANEDIT. In the following discussion of the commands to STANEDIT, the references are made to the command file.

<**match-key-dictionary**>: the match key dictionary defines each match key. There are separate match keys for each process. For example, there is a street address match key, a place match key, a name match key, and so on. Default match key dictionaries are provided with the system, but the user may modify the definitions, if needed, for specific applications.

<**classification-table**>: the classification tables allow STANEDIT to identify and classify keywords that may appear in the file. Examples of such keywords are the street types (ST, AV, BL), the directions (N, NW, S), titles (MR, DR), and so on. The classification table also provides standard abbreviations for each keyword. Default classification tables are provided, but the user can add additional entries or modify the tables as desired. There are separate tables for the separate processes: street address, place, name, and so on.

<**pattern-table**>: the pattern tables define the rules by which STANEDIT operates on the input fields. There is one pattern table for each process: address, place, name, and so on. Again, default rules are provided. The user may modify these rules as required.

Summary

Remember that the input to STANEDIT is a single record for interactive users containing information in free format fields, and the output from STANEDIT is the same record with match keys “prepended” (the match key becomes a prefix to the output record). These match keys are small records containing the input information with the contents properly standardized and separated into individual fields.

Getting started quickly

Despite the length of this manual and the many features of STANEDIT, a user can start using the program almost

immediately. To show how easy it is, suppose you have a file organized as follows:

The record type is ASCII with all of the records in the same format.

The record size is 256 characters.

The complete STANEDIT command file to process this would be:

```
RECORD 256
TYPE ASCII
INTERACTIVE
DEBUG
STANDARDIZE us_addr
OUTFILE us_addr.out
```

The **US_ADDR** process handles the U.S. street addresses such as **123 Main St.**

Finally, suppose that the above specifications were contained in a file named **us_addr.stn**. This file would be prepared using a text editor, such as Notepad. After a Windows program item for STANEDIT is added, clicking the program icon will display a WINSTAN window with a prompt for a command filename. Command files may be entered with or without the .stn extension.

Input file format specifications

Running STANEDIT

It is necessary to define both the format of the input file to be standardized by STANEDIT as well as what processes are desired. This is accomplished by means of a command file. This file must be prepared for each run of STANEDIT. It is a standard ASCII file that can be created and maintained with any standard text editor (for example, Notepad). To run STANEDIT, merely specify the name of the command file:

Prepare a command file using Notepad:

```
edit test.stn
```

Enter a command filename in the STANEDIT window:

```
Enter command file name: test.stn
```

In the above example, the command file was named test.stn, but any name can be used.

Preparing a STANEDIT command file

The STANEDIT command file consists of a number of commands or statements. Each command has a name followed by zero or more arguments. Arguments are separated from each other by one or more spaces. Only one command per line is allowed. Comments may be inserted anywhere following a semicolon. For example:

```
; This is a comment
```

```
RECORD 256 ; This is a command with one argument
```

Certain commands are required for every run, while others are optional. All optional commands have default values that are assumed if the command is not present. This simplifies the process of coding command files, but may produce unexpected results if the user is not aware of the default values.

The order of the commands is important. To ensure correct operation, the commands should be coded in the order presented in this section.

Output file name specification (OUTFILE)

The output file will contain the results of the standardizing, along with information about what patterns were matched by each input case. OUTFILE specification supplies the name of the output file to be processed. This file is created if it does not exist or deleted and created again if it existed previously. The OUTFILE specification has the following format:

```
OUTFILE <output-file-name>
```

where <output-file-name> is the name of the output file.

```
OUTFILE customer.std
```

```
OUTFILE C:\cust\customer.std
```

An OUTFILE command is required for each run of STANEDIT. Again, drive and path information may be included as part of the name.

Specification of record size (RECORD)

The RECORD command is used to specify the size of each record to be processed. This command is required for each run. If variable size records are present, this command should specify the size of the longest record. Do not include the carriage-return and line-feed characters in the record size. If there are multiple lines in a record, this size should specify the total length of the record (all lines), excluding the carriage-return line-feed characters that separate the lines. The RECORD command has the following format:

```
RECORD <record-size>
```

where <record-size> is the size of the record in characters.

For example, RECORD 200 indicates that the input record to be processed is 200 characters long.

The largest record that may be specified is 1,024 characters. ArcView specifies the record size to be 256 characters.

File type specification (TYPE)

The TYPE command is used to specify what type of file is being processed. The TYPE statement has the following format:

```
TYPE ASCII
    UNIX
    BINARY
```

ASCII, UNIX, or BINARY must be chosen.

ASCII: a standard PC text file. If the file can be printed properly with a TYPE or a PRINT statement, or if it can be read with a standard text editor, then it is an ASCII file. Such files have carriage-return and line-feed characters at the end of each line.

UNIX: a UNIX text file has only a line-feed character after each line. If printed or typed on a PC, the text will not appear properly left justified.

BINARY: a BINARY file consists of fixed length records with no carriage-return or line-feed separators between records. If printed or typed on a PC, it appears as one long stream of text. BINARY files may contain binary values (internal representations of numbers, for example). STANEDIT requires all fields to be processed to be in ASCII representation.

If the TYPE command is missing, ASCII is assumed on PC implementations, and UNIX is assumed on UNIX machines. If ASCII is specified on UNIX machines, then UNIX format is assumed.

ArcView assumes the type is ASCII.

Interactive processing (INTERACTIVE)

The INTERACTIVE command indicates that there is no input file. This is required when STANEDIT is used as part of an interactive process, or input records are not obtained directly by means of sequential read operations. This statement should be used in specifications for the test program STANEDIT.

Debugging patterns (DEBUG)

The DEBUG statement may be included in the specifications to test the operation of the pattern matching. Information about which patterns were matched and what data was moved to the match key is printed on the standard output.

The OUTFILE will contain the results of the standardizing along with information about what patterns were matched by each input case. An example specification file to debug the us_addr process is:

```
RECORD 256
TYPE ASCII
DEBUG
INTERACTIVE
STANDARDIZE us-addr
OUTFILE us_addr.out
```

Once the formats and types of files have been defined, the user must specify what processing is desired and what type of information each field or item contains. This is accomplished through one or more STANDARDIZE commands. These commands should directly follow the commands described in the previous section. The STANDARDIZE commands have the following format:

```
STANDARDIZE <process-name>
```

The process-name refers to the match key dictionary, classification table, and pattern file that are set up to handle a certain type of input.

Address components

A postal address is much more complex than an individual name. First of all, there are many more components and variations of formats. This discussion of address standardization begins with a definition of the components of an address and some basic concepts. However, you must remember that STANEDIT is a rule-based system, so the interpretation of fields, the definition of components, and the classification of keywords can all be modified by the user. Such modifications are important for international addresses and for handling regional problems or situations common to certain input file peculiarities.

This discussion of the components of an address is biased toward United States address styles. However, the international user should not skip this section since the explanation of the basic concepts is applicable in all cases. An understanding of these concepts is important for understanding the operation of the standardizer.

A postal address contains two basic parts:

MTCHADDR: Street address

PLACE: City, state, and postal code

The MTCHADDR process handles house number and street name information with an emphasis on using the output in matching applications. The PLACE process handles the city name, state name or abbreviation, and ZIP Code. A street address can be divided into a number of parts as follows:

HOUSENO: the house number or low number in a range

COORDHOUSE: coordinate house number or high number in range

HOUSESUFFIX: house number suffix

PREDIRECTION: street direction

PRETYPE: prefix type

STREETNAME: street name

SUFFIXTYPE: suffix type

SUFFIXDIRECTION: postdirection

UNITTYPE: type of unit

UNITVALUE: value of unit

RURAL ROUTE TYPE: rural route type

RURAL ROUTE NUMBER: rural route number

BOX TYPE: post office box or rural box type

BOX NUMBER: post office box or rural box number

The PLACE is divided into the following parts:

CITY: city name

STATE: state or province name

POSTCODE: postal code

These components can be illustrated with the following examples:

14637 LOCUSTWOOD LANE, SILVER SPRING, MD 20905

HOUSENO: 14637

STREETNAME: LOCUSTWOOD

SUFFIXTYPE: LANE

CITY: SILVER SPRING

STATE: MD

POSTALCODE: 20905

Addresses in Queens, New York, have coordinate house numbers. The 04 in the following example indicates the 100 block of the cross street:

1234-04 NORTHERN BLVD, QUEENS NY 10230

HOUSENO: 1234

COORDHOUSE: 04

STREETNAME: NORTHERN

SUFFIXTYPE: BLVD

The following subsections describe the components of an address in more detail.

House address components

The house address can be subdivided into a house number, a coordinate house number, and a house number suffix. The house number field is used to store the principal house number for the address. Coordinate house numbers, or a high house number range, are stored in the coordinate house number field. This is useful for cities, such as Queens, New York, where a coordinate house number indicates an intersecting street. Further, the coordinate house number field can be used for standardizing reference files, such as the U.S. Postal Service ZIP+4 files. The low address range can be placed in the principal house number field, and the high address range can be placed in the coordinate house number field.

The house number suffix field is for fractional addresses and for unit designations attached to the house number.

Consider the following examples:

123-04 House number = 123, Coordinate house number = 04

123 1/2 House number = 123, House suffix = 1/2

123-A House number = 123, House suffix = A

For files such as the U.S. Census Bureau TIGER files, there are four house numbers on each record. These represent the FROM LEFT, FROM RIGHT, TO LEFT, and TO RIGHT addresses on a street segment. Since these are already in standardized form, it is not necessary to use all of the ranges. The matching software can access the fields directly. Consequently, to obtain correct results, pick one number (for example, FROM LEFT), to ensure a standard style of address (that is, number, street name, and so on), and use all four original address number fields in the matcher run.

Street components

A typical street address has additional components attached to the street name. These are generally directions (NORTH, SOUTH) and types (STREET, AVENUE), but there can be fairly complex variations. Apartment number (unit) information is also considered a street component, since it is generally written on the same line of the address as the street—for example, 123 Main St, Apt 3-G.

There are two basic reasons for address standardization. The first is to support mailing to customers, clients, or members on a mailing list. In this case, it is important to preserve the order of the components. For example, in the two cases N MAIN ST and COLUMBIA RD NW, the first case has a direction (north), which usually precedes the street name. However, for the second case, the NW refers to a quadrant in the city of Washington, D.C., and usually follows the street type. It would be placed in the post direction field.

The second reason for address standardization is to support matching or searches in a database. In this case, it is less important to read the address properly and more important to avoid a large number of combinations of fields. For example, you would want N MAIN ST and MAIN ST N to be standardized in

the same way. The north should be placed in the direction field in both cases. This avoids having to compare the direction with both the direction and the postdirection fields, although the matching software can easily do this.

The same arguments apply to 3RD AVE and AVE 3. For mailing purposes, the first case has a type of AVE and the second case has a prefix type of AVE. For matching purposes, it would be better to code AVE as the type in both cases.

Since STANEDIT is a rule-based system, the rules can support either objective. If a file is to be used both for mailing and matching, it is best to give priority to mailing clarity. These discussions give priority to mailing purposes, since this only makes the match rules slightly more complex.

Notice that there are five basic street components. Predirection and pretype occur before the street name. Suffix direction (postdirection) and Suffix type occur after the street name. Consider the following examples:

Coordinate addresses, found in Salt Lake City, have a format as follows:

123 WEST 234 SOUTH

Since it is important to have a street name for matching purposes, the following parsing is recommended:

123: house number

W: predirection

234: street name

S: suffix direction

This is better than using the coordinate house number fields for the address numbers and ending up with a blank street name.

Multiunits

The term **unit** refers to an apartment, suite, condo, or other dwelling or office area. A **multiunit** is where more than one unit shares a common house number. Multiunits have a **type** (or the kind of unit) and a **value** (the name or designation) of the unit. Consider the following examples in the table below:

Post office boxes

Post office boxes can coexist with a standard street address. For example, an address could be: 123 MAIN ST P.O. BOX 1234.

In this case, the box information is redundant and should be moved into the post office box field of the match key.

For cases where the post office box is the entire street address—for example, PO BOX 456—this information is also placed in the post office box fields. The box type field will contain the word BOX, and the box value field will contain 456.

Rural addresses

Rural addresses are divided into rural routes and highway contracts (star routes). The exact rural location is indicated by a box number. The rural route type and value fields are used to store the route information, and the box type and value fields are used to store the box information. Examples:

RT3, BOX 15

Rural route type = RR

Rural route value = 3

Box type = BOX

Box number = 15

Highway contracts have a route type of HC. Star routes also have a route type of HC.

Highways, such as US RT 101, have a route type of USRT and a value of 101.

Place components

The place consists of a city name, a state or province, and a postal code. In the United States, there are 5 and 9 digit ZIP Codes. In other countries, the postal codes are often combinations of numeric and alphabetic characters in a certain recognizable configuration.

The spellings of place names must be standardized so that there is a higher probability of matching correctly. For example, L.A. should be expanded to LOS ANGELES, and BOST should be expanded to BOSTON.

State names should be converted to the proper two-character FIPS abbreviations. For example, WEST VIRGINIA should be converted to WV.

Standardizing addresses (MTCHADDR)

The fields for each process should be specified in the order that the input would normally be read:

123 Columns 10–16

N MAIN ST Columns 20–44

SILVER SPRING Columns 60–79

MD Columns 80–81

20905 Columns 82–86

The specification for this file would be:

STANDARDIZE MTCHADDR

Specifying parsing parameters (SEPLIST and STRIPLIST)

There are default rules that define what constitutes a token or operand as defined in the pattern file. For example, periods are stripped from the input records as if they never existed. Thus, N.W. and NW are both considered to be the single word NW. Hyphens separate words and are considered words in themselves. For example, 123-456 is three words (tokens): 123, the hyphen (-), and 456.

Spaces are both stripped and separate tokens. For example, 123 Main St is three tokens: 123, Main, and St.

You can override the default assumptions by specifying a SEPLIST and a STRIPLIST. Any character in the STRIPLIST is removed as if it never existed. Any character in the SEPLIST is used to separate tokens. Any character that is in both lists will separate a token but will not appear as a token. The best example of this is a space. One or more spaces will be stripped, but the space indicates where one word ends and another begins.

If a STRIPLIST and a SEPLIST are desired, they must immediately follow the STANDARDIZE statement to which they apply. The characters in the list must be enclosed in quotation marks. The quotation mark itself may not be in either list:

STANDARDIZE PLACE

STRIPLIST “-”

SEPLIST “,”

In this example, the space is in both lists. Hyphens are stripped so that, for example, STRATFORD-ON-AVON is considered to be STRATFORDONAVON. The comma separates a token so that the city name and state can be found (SALT LAKE CITY, UTAH). The space, as always, appears in both lists. Any other special characters will be classified as a special type.

Each process may have its own lists. If no list is coded for a process, then the default lists are used.

When overriding the default SEPLIST and STRIPLIST, care must be taken not to cause collisions with the predefined class meanings. For instance, ^ is the numeric class specifier. If this character is added to the SEPLIST and not to the STRIPLIST, then a token consisting of ^ is given the class of ^. This token would then match to a numeric class (^) in a pattern file.

In ArcView, the default **STRIPLIST** is “.,\’;:””, and the default **SEPLIST** is “()-/,#&:;””

The match key

This section begins a discussion of the more advanced features of STANEDIT. Users who are satisfied with the operation of the program as it is supplied do not need to read further. However, those users wanting to obtain optimal performance or those users outside of the United States may wish to modify tables, match key definitions, patterns, and so on. The remainder of this appendix presents these topics.

Introduction to match keys

Each standardization process (MTCHADDR, PLACE, or any user-defined process) is defined by three entities:

- A match key definition
- A classification table
- A pattern file

These entities will be presented in the following sections.

The output from the standardization process is a match key. This is a small record with fixed length fields that receive the components of an address, a name, and so on. For example, the given name of an individual should always be placed in the given name field of the match key, regardless of where it was recognized in the input record. Street types should have standardized abbreviations placed in the match key. For example, ST, STR, and STREET should all be standardized to ST.

The match key is defined in a match key dictionary. There is a separate match key for each of the processes.

The match key dictionary for each process is contained in a file named either PLACE.DCT (PLACE dictionary file) or MTCHADDR.DCT (MTCHADDR dictionary file).

The dictionaries are ASCII files that can be maintained with a standard text editor. Each line of the dictionary represents a field of the match key. The format of a dictionary line is:

```
<field-identifier> <field-type> <field-length> <missing> [ ;  
<comments>]
```

The field identifier is a two-character field name (case-insensitive) that must be unique over all the dictionaries defined.

The field type defines how information is to be placed in the field. The following are the only possible values, and one must be chosen for each field defined:

N or C must be coded to indicate whether the field is a character field or a numeric field. Numeric fields are right-justified and filled with leading blanks in the key, and character fields are left-justified and filled with trailing blanks.

NS may be coded to indicate that leading zeros should be stripped from a numeric field. For example, house numbers would be defined with type NS and ZIP Codes would be defined with type N, since leading zeros matter with ZIP Codes but not with house numbers.

M indicates mixed alphas and numerics. This type causes numeric values to be right-justified in the fields, and alpha values to be left-justified. Leading zeros, if present, are retained. This type helps retain the proper sort order for these fields. The U.S. Postal Service uses this type of field for house numbers and apartment numbers. For example, in a four-character type M field, 102 becomes b102, and A3 becomes A3bb (where b represents a space [blank]).

MN indicates mixed name. It is generally used for representing street names in sort order. Field values beginning with an alphabetic character are left justified. Field values beginning with a number are indented as if the number were a separate three-character field. For example, the following list demonstrates the correct sorting order:

```
MAIN  
CHERRYHILL
```

bb2ND

b13TH

l23RD

Notice that single-digit numbers are indented two spaces, two-digit numbers are indented one space, and three-digit numbers and alphabetic characters are not indented. This keeps numbers in a proper sort order. 2ND precedes 13TH. The U.S. Postal Service uses this type of field for street names in the ZIP+4 files.

The third operand is the field length in characters.

The fourth operand is a missing value identifier. This is included for compatibility with the interactive matching library. You should just code an X for this operand.

Optional comments may follow a semicolon. For example:

HN	N	7	X ; House number
SN	MN	25	X ; Street name
MV	M	6	X ; Multiunit value
CT	C	20	X ; City name

This sample key defines four fields. It creates a fixed record 48 characters long. It contains a house number field, identified by the abbreviation HN; a street name field, SN; a multiunit value field, MV; and a city field, CT. The house number is numeric, the street name is mixed-name, the multiunit value is mixed, and the city name is character. Notice the missing value operand (X).

The first line of a match key dictionary should be the following:

```
\FORMAT\ SORT=N
```

This line prevents the table from being sorted on field identifier. The fields in the match key will be in the same order that they are defined in the dictionary. There can't be any comments preceding this line.

The following subsections present the match key dictionaries for the processes supplied with the system.

Name match key

The NAME process is supplied with the following default match key:

```
\FORMAT\ SORT=N
```

```
;
```

```
; Name match key dictionary
```

```
;
```

```
TL   C           4       X ; Title (Mr., Mrs., Dr., etc.)
```

```
FN   C           16      X ; First name
```

```
MN   C           16      X ; Middle name
```

```
LN   C           16      X ; Last name
```

```
RK   C           4       X ; Rank (Jr., Sr., and so on)
```

```
XN   C           4       X ; Soundex of last name
```

To see how fields would be moved to the match key, consider the following example:

```
MR. JOHN CHARLES THOMAS, JR.
```

```
TL = MR
```

```
FN = JOHN
```

```
MN = CHARLES
```

```
LN = THOMAS
```

```
RK = JR
```

```
XN = T530
```

Soundex is a phonetic encoding scheme that is used to aid searching despite spelling errors. For example, TOMAS, THOMIS, and TOMS, would all have a Soundex code of T530.

You may change any of the field lengths as needed. If other changes are made (new fields added, fields are deleted, and so on) you must make appropriate changes to the pattern files. Remember that the two-character field names must be unique across all processes.

Place match key

The PLACE process is supplied with the following default match key:

```
\FORMAT\SORT=N
```

```
;
```

```
; Place match key dictionary
```

```
;
```

```
CT C 25 X ; City name
```

```
XC C 4 X ; Soundex of City name
```

```
SA C 2 X ; State Abbreviation
```

```
ZP N 5 X ; ZIP Code
```

```
Z4 N 4 X ; ZIP+4 add-on Code
```

The state abbreviation is the two-character standard USPS code for the state name. There are two ZIP Code fields, comprising both parts of a ZIP+4 Code. If there is no four-digit add-on code, then this field will be blank.

Consider the following example of standardized place information:

```
BERKELEY SPRINGS, WVA 12345-6789
```

```
CT=BERKELEY SPRINGS
```

```
XC=B624
```

```
SA = WV
```

```
ZP = 12345
```

```
Z4 = 6789
```

You can change field lengths, except for the Soundex fields and the state abbreviation. If the state abbreviation or postal code fields are different, then appropriate changes must be made to the pattern files.

Street address match key

The MTCHADDR process is supplied with the following default match key:

```
\FORMAT\SORT=N
```

```
;
```

```
; Street address match key
```

```
;
```

```
HNN 8 X ; House Number
```

```
CHN 8 X ; Coordinate house number
```

```
HSC 4 X ; House Number Suffix "A", "1/2", and so on
```

```
PDC 2 X ; Predirection
```

```
PT C 4 X ; Pretype
```

```
SN C 28 X ; Street Name
```

```
ST C 4 X ; Suffix type
```

```
SD C 2 X ; Suffix direction
```

```
UTC 4 X ; Multiunit type
```

```
UVC 10 X ; Multiunit value
```

```
XSC 4 X ; Soundex of Street Name
```

```
XRC 4 X ; Reverse Soundex of street name
```

Consider this example of an address:

1234-04 1/2 NORTH CEDAR KNOLLS LN EAST, SUITE 560-A

HN = 1234

CH = 04

HS = 1/2

PD = N

PT =

SN = CEDAR KNOLLS

ST = LN

SD = E

UT = STE

UV = 560-A

XS = C360

XR = S452

You may change field lengths (except for Soundex fields) as desired. However, deleting and adding fields will require changes to the pattern files.

The reverse Soundex code is used to facilitate address matching. It is a Soundex code of the street name spelled backwards. Thus, the two Soundex codes are developed from **CEDAR KNOLLS** and **SLLONK RADEC**.

It should be clear from the discussion in the previous section that each standardization process requires a match key definition, a classification table, and a pattern file. The next section describes the classification table.

The classification table

Classification table format

The classification table is a standard ASCII file with one line per entry. Comments can follow a semicolon (;). Each entry contains the keyword, the standardized abbreviation, the keyword class, and the optional comparison threshold.

Consider the following examples of classification table entries:

S	S	D	
SO	S	D	
WEST	W	D	
WST	W	D	
W	W	D	
EAST	E	D	
E	E	D	
NORTHWEST	NW	D	800.0
NW	NW	D	
ROUTE	RT	T	800.0
RTE	RT	T	

These entries show classification of several directions and the street type route. The first entry in a line is how the word would be spelled in the input file. The second entry is the standard abbreviation that should be used for that keyword in the match key. For example, regardless of whether west was coded as WEST, WST, or W, the standard abbreviation would always be W. The W would appear in the appropriate field in the match key. The third entry on the line specifies the class that should be assigned to this word. These classes can be assigned by the person constructing the tables. For example, the class D was assigned to indicate direction and the class T was assigned to indicate street type. Thus, all variations of South, West,

NorthEast, and so on are assigned a class of D to indicate that they are directions. These classes are used in the pattern files to specify the rules that will be used to interpret the elements of an address.

The standardized abbreviation is limited to 25 characters in length. The keyword must be a single word. Multiple or compound words are considered separator keywords.

Threshold weights

The optional number at the end of each entry is a weight threshold that can specify the degree of uncertainty that can be tolerated in the spelling of the keyword. An information-theoretic string comparator is used that can take into account phonetic errors; random insertion, deletion, and replacement of characters; and transpositions of characters. The score is weighted by the length of the word, since small errors in long words are less serious than errors in short words. In fact, the threshold should be omitted for short words since errors generally cannot be tolerated. The numeric score operates roughly as follows:

900: exact match

800: strings are almost certainly the same

750: strings are probably the same

700: strings are probably different

Lower numbers tolerate more differences between the strings.

Special classes

You can assign any single character class desired to a keyword. The classes may be from A to Z. A special class (0) is available to indicate a null (or noise) word. Null words are skipped in the pattern matching process. For example:

OF 0 0

The standard abbreviation can be any value desired in this case since it will not be used.

Address classification tables

Each process requires a classification table. Each table operates in an identical manner. It doesn't matter whether addresses, names, companies, or places are being processed. Each one requires a classification table. The person designing the tables and rules can decide upon the classification schemes. However, this section will describe the schemes used for the classification table supplied with the program.

The following classification types are used:

D	Direction	East, West, Southwest
T	Street type	Avenue, Street, Place
M	Multunit	Apt, #, Suite, Room
B	Box	P.O. Box
0	Null	Of, the, for
X	Descriptions	Lot
O	Ordinals	SECOND, THIRD
C	Cardinals	SIXTEEN, THREE
N	Number	FIFTY, SIXTY
S	Spanish	LA
Q	Post	
R	Office	

The treatment of numbers requires some explanation. Ordinal numbers are given a class of O. Cardinal numbers are given a class of C. However, beginning with 20, the numbers 20, 30, 40, and so on are prefixes and can either be ordinal (twenty-first) or cardinal (twenty). Therefore, they are given the class N to

distinguish these cases. The class S is used to handle the Spanish article LA, which is also an abbreviation for LANE. Thus, LA CIENEGA is not a street type followed by a single word, but a single street name. Finally, the classes Q and R are used for post office boxes when the single letter abbreviations P O BOX are used. Both P and O can also be street names when they stand alone.

Place classification table

The following classifications are used for the place classification table:

S	State	Maryland, Wyoming
P	Prefix	New, North, South
Q	Suffix	Island (in Rhode Island)

Reclassification of tokens

If more than one process has the same format specified, then the first process receives the input record. The tokens are classified according to the classification table for process 1. Process 1 may filter out information by retyping tokens to null (0). Process 2 will receive the token table from process 1 and first reclassify it according to the classification table for process 2.

This is important when the exact formats of the input record are unknown. For example, if place information could coexist with street address information in the same field, then the PLACE process could filter out the city, state, and ZIP information, leaving the street address information for the MTCHADDR process.

As many processes as desired may share the token table. The fact that this occurs is largely transparent to the end user since it is driven by detecting that the input formats for more than one process are the same.

The pattern file contains the rules by which the standardization is accomplished. The major power of STANEDIT rests with its ability to process powerful and flexible rules. This appendix presents an introduction to the concepts of pattern matching and the reasons that such matching is required to obtain correct standardization in all instances.

The pattern rules

Pattern matching

If all elements of an address were uniquely identified by keywords, then address standardization would be easy. For example, 123 N Main St is obviously easy and not subject to any ambiguity. The first numeric is the house number. This is followed by a direction, uniquely identified by the keyword N, followed by an unknown word MAIN, followed by a street type, ST. Most addresses fall into this pattern with minor variations:

123 E Maine Av

3456 No Cherry Hill Road

123 South Elm Place

These addresses all fit into the pattern:

Numeric

Direction

Unknown word or words

Street type

The first numeric should be interpreted as the house number and moved to the house number field of the match key {HN}. The direction should be moved to the predirection field {PD}, the street name words should be moved to the street name {SN} fields, and the street type should be moved to the {ST} field. Notice that these are the two character field names defined in the match key dictionary. The braces indicate that the reference is to a match key field. Given the pattern, here are the actions:

Numeric: {HN}

Direction: {PD}

Unknown word or words: {SN}

Street type: {ST}

Tokenization and classification

Standardization begins by separating all of the elements of the address into tokens. Each token is either a word or a number, or a mix of the two, separated by one or more spaces. At the same time the tokens are formed, each token is classified by looking to see if it is in the classification table. If it is, it is given the class indicated by the table. If it isn't, it's given one of the following classes:

^: Numeric

?: One or more consecutive unknown alpha words

>: Leading numeric

<: Leading alphabetic

@: Complex mixed

~: Special

0: Null

-: Hyphen

/: Slash

&: Ampersand

#: Number sign

(: Left parenthesis

): Right parenthesis

A numeric token contains all digits, for example, 1234.

An unknown token contains one or more unknown words, for example, Cherry Hill.

A leading numeric contains numbers followed by one or more letters, for example, 123A.

A leading alphabetic contains letters followed by one or more numbers, for example, A3.

A complex mixed token is a mixture of alphas and numerics that do not fit into the above classes, for example, 123A45, ABC345TR.

A special token contains special characters that are not generally encountered in addresses. These include !, @, ~, %, and so on.

A null token is any word that is to be considered noise. These words may appear in the classification table and are given a type of zero. Similarly, actions can convert normal tokens into null tokens.

The standard address forms:

123 E Maine Av

3456 No Cherry Hill Road

123 South Elm Place

would be tokenized and classified as follows:

123	^	Numeric
No	D	Direction
Cherry Hill	?	Unknown words
Road	T	Street type

The pattern represented by this address can be coded as:

```
^|D|?|T
```

The vertical lines separate the operands of a pattern. All of the addresses above will match this pattern. The classification of D comes from the classification table. This has entries indicating that No, East, E, NW, and so on, are all given a class of D to indicate that they generally represent directions. Similarly, the classification of T is given to entries in the table representing street types (Road, St, Place, and so on).

Patterns and actions

The pattern file consists of a series of patterns and associated actions. After the input record is separated into tokens and classified, the patterns are executed in the order they appear in the pattern file. A pattern either does or doesn't match the input record. If it matches, then the actions associated with the pattern are executed. If it doesn't, the actions are skipped. In either case, processing continues with the next pattern in the file.

The pattern file is a standard ASCII file that can be created or updated using any standard text editor. It has the following general format:

```
\POST_START
<post-execution actions>
\POST_END
<pattern>
<actions>
<pattern>
<actions>
```

There is one special section in the pattern table—the poststart actions. The poststart actions are those actions that should be executed after the pattern matching process is finished for the input record. An example of a postaction is computing Soundex codes for street names. The special section is optional. If omitted, then the header and trailer lines should also be omitted.

Other than the special section, the pattern file consists of sets of patterns and associated actions. The pattern requires one line. The actions are coded one action per line. The next pattern may start on the following line.

Blank lines may be used freely to increase readability. For example, it is suggested that blank lines or comments separate one pattern/action set from another.

Comments are indicated by a semicolon. All characters following a semicolon (;) are considered to be comments. An entire line may be a comment line by specifying a semicolon as the first nonblank character. For example:

```
;  
; This is a standard address pattern  
;  
^|?|T ; 123 Main St
```

As an illustration of the pattern format, consider postactions of computing a Soundex code for street name and processing patterns to handle:

```
    123 N Main St  
    123 Main St  
\POST_START  
SOUNDEX {SN} {XS}  
\POST_END  
^|D|?|T           ; 123 N Main St  
COPY [1] {HN}     ; Copy House number (123)  
COPY_A [2] {PD}   ; Copy direction (N)  
COPY [3] {SN}     ; Copy street name (Main)  
COPY [4] {ST}     ; Copy street type (St)  
EXIT  
^|?|T  
COPY [1] {HN}
```

```
COPY [2] {SN}  
COPY_A [3] {ST}  
EXIT
```

Notice that this example pattern file has a postsection that computes the Soundex of the street name (in match key field {SN}) and moves the result to the {XS} match key field.

The first pattern matches a numeric, followed by a direction, followed by one or more unknown words, followed by a street type (such as 123 N Main St). The associated actions are to copy operand [1] (numeric) to the {HN} house number field of the match key. Copy the standard abbreviation of operand [2] to the {PD} prefix direction field of the match key. Copy the unknown word or words in operand [3] to the {SN} street name field of the match key. Copy the standard abbreviation of the fourth operand to the {ST} street type field of the match key. Exit the pattern program. A blank line indicates the end of the actions for the pattern.

The second pattern/action set is similar to the first, except that it handles cases like 123 Main St. If there is no match on the first pattern, then the next pattern in sequence is attempted.

Pattern format summary

This section discusses the basic format of patterns and how various elements of the standardization process can be referenced.

A pattern consists of one or more operands. Each operand is separated by a vertical line. For example, the pattern ^|D|?|T has four operands. These are referred to in actions as [1], [2], [3], and [4]. The unknown class (?) refers to one or more consecutive unknown alphabetic words. This simplifies pattern construction, since names like Main, Cherry Hill, and Martin Luther King will all match to a single ? operand.

Spaces may separate operands if desired. For example, `^|D|?|T` is equivalent to `^| D | ? | T`.

Comments may follow a semicolon:

```
;
; Process standard addresses
;
^|D|?|T ; 123 N Main St
```

Match key fields may be referred to by enclosing the two character match key name in braces. For example, `{SN}` refers to the street name field (SN) of the match key.

Pattern matching stops after the first match is encountered. For example, in an input record like 123 Main St & 456 Hill Place, the pattern `^|?|T` matches to 123 Main St and not to 456 Hill Place.

The simplest patterns consist only of classification types. For example:

```
^|D|?|T
```

These are straightforward and do not require much further explanation. Remember that hyphens and slashes may be present in the patterns. For example:

```
123-45 matches to ^|-|^
```

```
123 1/2 matches to ^|^|/|^
```

Unconditional patterns

This section presents information on unconditional pattern matching. These patterns are not sensitive to individual values. They are the simplest to code and the most general. Conditions can be specified to cause patterns to match only under specified circumstances. Conditions are discussed in the next section, Conditional patterns.

Simple pattern classes

This section describes the simple pattern classes. Any of the simple classes may appear in a pattern specified in the Pattern Rule file. These classes are slightly different from the classes assigned to a string when the input record is read. This is because several forms can match to a single input token. In other words, this is a pattern-matching language. The simple pattern classes are:

A-Z: Classes supplied by user from classification table

^: Numeric

?: One or more consecutive unknown alpha words

+: A single alphabetic word

&: A single token of any class

>: Leading numeric

<: Leading alphabetic

@: Complex mixed

~: Special

-: Hyphen

/: Slash

\&: Ampersand

\#: Number sign

\(: Left parenthesis

\): Right parenthesis

Notice that the null class (0) is not included in this list. The null class is used either in the classification table or in the RETYPE action to make a token null. However, since it never matches to anything, it would never be used in a pattern.

The classes A through Z correspond to classes coded in the classification table. For example, if APARTMENT is given the class of M in the classification table, then a simple pattern of M will match.

The class ^ represents a single number. For example, 123 will match. Also, 123.456 will match since periods are filtered out of the input stream. The number 1,230 is three tokens: the number 1, a comma, and the number 230.

The class ? matches to one or more consecutive alphabetic words. For example, MAIN, CHERRY HILL, and SATSUMA PLUM TREE HILL all match to a single ? operand, providing none of the aforementioned words are in the classification table for the process. This is useful for street names where a multiword street name should be treated identically to a single word street name.

A single alphabetic word can be matched with a + class. This would match the first word in each example. This is useful for separating the parts of an unknown string. For example, in a name like JOHN QUINCY JONES, the individual words can be copied to a match key with first name {FN}, middle name {MN}, and last name {LN}, as follows:

```
+|+|+
COPY [1] {FN}
COPY [2] {MN}
COPY [3] {LN}
```


A leading numeric class (>) is useful for address suffixes. For example, a house number like 123A MAIN ST can be matched as follows:

```
>|?|T
```

The leading alpha class (<) would match to A123, ABC123, and so on.

The complex class (@) matches those tokens having a complex mixture of alphas and numerics. Some examples of this include A123B and 345BCD789.

The special class (~) is used for tokens containing leading special punctuation marks not mentioned in the list. This includes dollar sign, percent sign, quote marks, and so on. For example, \$HELLO is a special class token.

The hyphen (-), slash (/), number sign (\#), left and right parentheses (\(and \)), and ampersand (\&) are all special single character classes. For example, an address range like 123-127 would match the following pattern:

```
^|-|^
```

The slash is useful for fractional addresses, such as 123 1/2 MAIN ST. This matches to:

```
^|^\|/|^ \ 123 1/2
```

Number signs are useful for apartment numbers, and ampersands are used for coding street intersections (1ST & MAIN ST).

Since the classes for number sign, ampersand, and left and right parenthesis may interfere with the syntax of the pattern tables, the parentheses, number sign (#), or ampersand must be preceded by a backslash (\). The backslash is called an escape character since it signals the presence of a nonstandard following character.

For example, to filter out parenthetical remarks such as:

(see Joe, Room 202)

you could code:

```
\(|**|\)  
RETYPE[1]0  
RETYPE[2]0  
RETYPE[3]0
```

This removes the parentheses and the contents of the parenthetical remark.

A single token of any class may be represented by the ampersand (&). For example, a pattern to match to a single word following **APARTMENT, SUITE**, and so on, would be:

```
* M | &
```

This would match:

```
APARTMENT 3G  
SUITEA301  
ROOM45
```

However, in a case such as:

```
BLDG3-5
```

only BLDG 3 would be picked up by this pattern.

Care must be taken when specifying SEPLIST and STRIPLIST entries. For example, for the ampersand (\&) to be recognized as a single token, it must be in the SEPLIST but not in the STRIPLIST. This is true of the default lists.

End of field specifier (\$)

The \$ specifier does not actually match any real token, but matches to the end of the field. Imagine that after each string

there is an end-of-field marker. The \$ matches to this marker. Do not confuse the fields mentioned here with the fields specified in the STANDARDIZE statements. The input processor concatenates all the formats together with implied spaces between each format to create one large field. It is this field that is tokenized and compared to the patterns coded.

The \$ specifier is used to make sure no tokens are left in the field after the pattern. For example, if you want to match to city, state, and ZIP Code, you need to make sure that no tokens follow the ZIP Code. For example, the pattern *^ would match any numeric token—and in particular, the first numeric token encountered—but the pattern *^| \$ would only match a number if it were the very last token. Thus, this would automatically pick up the ZIP Code and avoid matching to house numbers, numeric streets, and so on. The asterisk (*) is a positioning specifier and means that the pattern will be searched until there is a match or the entire pattern is scanned.

Subfield classes (1 to 9, -1 to -9)

The classes 1 to 9 and -1 to -9 are used to pick off individual words of a ? string. 1 represents the first word, 2 represents the second, -1 represents the last word, -2 represents the next to last word, and so on. If the referenced word doesn't exist, then the pattern doesn't match. Suppose you were processing company names and you only wanted the first word. A company name such as JOHNSON CONTROL CORP would match to the following (assume CORP is in the classification table as a type C):

```
?|C \JOHNSON CONTROL is operand [1], CORP is operand [2]
```

```
1|C \JOHNSON is operand [1], CORP is operand [2]
```

```
2|C \CONTROL is operand [1], CORP is operand [2]
```

```
-1|C \CONTROL is operand [1], CORP is operand [2]
```

```
-2|C \JOHNSON is operand [1], CORP is operand [2]
```

```
++|C \JOHNSON is operand [1], CONTROL is [2], CORP is [3]
```

Another good example of using the subfield operand is for processing address suffixes. Suppose an example input address was 123-A MAIN ST.

A pattern such as:

```
^|-|?
```

would match as follows:

```
[1]=123
```

```
[2]=-
```

```
[3]=A MAIN
```

This isn't the desired result, since both **A** and **MAIN** are part of the unknown token. However, a pattern like:

```
^|-|1
```

would match as follows:

```
[1]=123
```

```
[2]=-
```

```
[3]=A
```

The following pattern and actions could be used to process the address suffix:

```
^|-|1 \123-A main st
```

```
COPY [3] {HS}
```

```
RETYPE [3]0
```

This pattern “filters” out the house address suffix by copying it to the match key and retyping it to be a null operand.

Single alpha classes (+) can be combined with subfield classes. For example, consider a series of consecutive unknown tokens,

such as CHERRY HILL SANDS. The pattern +|-1 would match as follows: the + would match to the word CHERRY. This leaves the string HILL SANDS as the ? token. The -1 would match to SANDS. Thus, operand [1] is CHERRY and operand [2] is SANDS.

Subfield ranges (beg:end)

The preceding section discussed how single subfields of an unknown series of words can be represented. You can also specify a range of unknown words. The format is (beg:end). For example:

(1:3) specifies words 1 to 3 of an unknown string.

(-3:-1) specifies the third from the last to the last word of an unknown string.

(1:-1) specifies the first to last word (using ? is more efficient).

So, in the address 123 - A B Main St, the pattern:

```
^|-|(1:2)
```

```
COPY [3] {HS}
```

```
RETYPE[3]0
```

results in AB being moved to the {HS} match key field.

Universal class (**)

The class (**) matches all tokens. For example, a pattern of ** would match 123 MAIN ST, and will also match 123 MAIN ST, LOS ANGELES, CA 90016, and so on.

The universal class can be combined with other operands to restrict the tokens grabbed by the class.

For example, **|T matches to all tokens before the type, which may be no tokens, then the type. Thus, 123 N Main St would

match with operand [1] being 123 N Main and operand [2] being St.

It is important to note that no tokens are required to precede the type. Thus, Avenue also matches to this pattern, with operand [1] being null.

In a pattern such as ^|**|T, the ** refers to all tokens between the numeric and the type.

A range of tokens may be specified for a ** operand. For example, ^|**(1:2)|T matches a numeric, followed by at least two nonstreet-type tokens, followed by a street type. Operand [2] would consist of exactly two nonstreet-type tokens. This matches 123 Cherry Hill St, but not 123 Main St, since only one token follows the number. Ranges, such as (1:1) to take any single token and (1:-1) to take the first to the last token, are permitted. However, a pattern such as **(1:1) results in much slower processing time than the equivalent & to match to any single token.

No conditional values or expressions are permitted for operands with **.

Floating positioning specifier (*)

In the patterns discussed prior to this section, the pattern had to match the first token in the field. For example, ?|T would match MAIN ST and CHERRY HILL RD, but wouldn't match 123 MAIN ST since a number is the first token.

Positioning specifiers are used to modify the positioning of the pattern matching. Floating specifiers are used to scan the input field for a particular pattern. The asterisk (*) is used to indicate that the class immediately following is a floating class. Suppose you wanted to scan for an apartment number, process it, then retype it to nulls. Retyping it allows patterns further down to process the address without the noise of having an apartment number in the string.

Addresses such as 123 Main St Apt 34 or 123 # 5 Main St match to the pattern * M | ^. The * M results in examining all tokens until a multiunit class token is found. Note that a pattern of M | ^ would not match since the first token is the house number. The asterisk indicates that the pattern is to be scanned forward until a class M token followed by a numeric is found. The following pattern and actions can be used to filter out the apartment number information:

```
*M | ^  
COPY_A [1] {UT}  
COPY [2] {UV}  
RETYPE [1] 0  
RETYPE [2] 0
```

Retyping these fields to null essentially removes them from consideration by any patterns that appear further down the file of patterns.

Floating positioning specifiers avoid the problem of having to enumerate all combinations of possibilities. For example, without floating patterns you would need individual patterns to handle cases such as:

```
123 Main St Apt 5  
123 N Main St Apt 5  
123 N Main Apt 5
```

With floating patterns, all of these addresses can be processed with the single pattern * M | ^.

Floating positioning specifiers operate by scanning each input token until a match is found. For example, in 123 N Main Apt 5, the 123 is tested. There is no match to * M. Then the N is tested. There is no match. Then Apt is tested. This matches the M. The token following the matched token is then tested. If all operands

match properly, the pattern matches, but if not, the scanner is advanced one token and the process is repeated. This is similar to moving a template across the input string. If the template works, the process is done; otherwise, the template is advanced to the next word. Note that there can only be one match to a pattern in an input string. After the actions are processed, control goes to the next pattern, even though there may be other matches on the line.

The asterisk must be followed by a class. For example, * M, * ?, and * ^ are all valid operands with a floating positioning specifier followed by a standard class.

There may be more than one floating positioning specifier in a pattern. For example, * ^ | ? | * T will match to JOHN DOE 123 CHERRY HILL NORTH RD. Operand [1] is 123. Operand [2] is CHERRY HILL. Operand [3] is RD.

Reverse floating positioning specifier (#)

The reverse floating positioning specifier, signaled by a number sign (#), is similar to the floating positioning specifier (*) except that scanning proceeds from right to left instead of from left to right. This is useful for removing items that would appear at the end of a field, such as a ZIP Code, state name, apartment designation, and so on.

The reverse floating positioning specifier must only appear in the first operand of a pattern, since it is used to position the pattern.

Suppose you wish to find a ZIP Code. Assume that state names were given class S. The pattern:

```
#S | ^
```

would scan from right to left for a state name followed by a number. This would work for input strings such as:

California 45 Products, Phoenix Arizona 12345 Dept 45

The right-to-left scan positions the pattern to Arizona. The number following causes a match to the pattern. If no match was found, scanning would continue to the left until a state followed by a number was found. If you were limited to the standard left-right floating positioning specifier (*S | ^), then the California 45 would be incorrectly interpreted as a state name and ZIP Code.

Fixed position specifier (%)

Sometimes it is necessary to position the pattern matching at a particular operand in the input string. This is handled by the %n fixed position specifier:

%1: matches to the first token

%2: matches to the second token

%-1: matches to the last token

%-2: matches to the second from last token

The positions can be qualified by following the %n with a token type:

%2^: matches to the second numeric token

%-1^: matches to the last numeric token

%3T: matches to the third type token

%2?: matches to the second set composed of at least two contiguous alphabetic unknown tokens

The fixed position specifier (%) is only permitted as the first operand of a pattern, since it is used to position the patterns. The fixed positions treat each token according to its type. For example, consider the input field:

John Doe 123 Martin Luther St Salt Lake.

%2? matches to the second string of consecutive alpha words (Martin Luther)

%-1? matches to the last string of consecutive alpha words (Salt Lake)

%1 1 matches to the first string, first word (John)

%1 2 matches to the first string, second word (Doe)

%2 1: matches to the second string first word (Martin)

%-2 -1: matches to the next to last string, last word (Luther)

%3+: matches to the third single alpha word (Martin)

%-1+: matches to the last single alpha word (Lake)

The position specifier does not continue scanning if a pattern fails to match (unlike * and #).

Thus, a pattern such as %3^ | D would match the 789 S in the string 123 A 456 B 789 S but would not match 123 A 456 B 789 C 124 S. This is because the third number (789) is not followed by a direction.

Negation class qualifier (!)

The exclamation point (!) is used to indicate NOT. Thus, !T means not Type and will match to any token except a street type. !? matches to any nonalpha unknown token. For example, * M | !T matches to SUITE 3, APT GROUND, but not to SUITE STREET.

Another example of the negation class qualifier comes if you wish to consider RT 123 to be the street name only if there is no unknown word following (such as RT 123 MAIN ST).

A pattern such as *T | ^ | !? matches to RT 123 but not to RT 123 MAIN ST, since an unknown alpha follows the numeric operand.

The negation class may be combined with the floating class (*) only at the beginning of a pattern. For example, when processing street addresses, you want to properly expand ST to SAINT when appropriate. Thus, ST CHARLES ST would be changed to

SAINT CHARLES ST, but MAIN ST REAR APT should not be changed to MAIN SAINT REAR APT. The pattern/action set:

```
*!?!T=ST|+
```

```
RETYPE [2]? "SAINT"
```

accomplishes this goal, since it requires that no unknown precede the value ST.

Conditional patterns

A defect in early attempts at address standardization was the inability to attach conditions to pattern matching. For example, ST CHARLES and AVENUE OF THE AMERICAS both match to T | ?, since both ST and AVENUE are street types. However, in the first case, the ST refers to SAINT, and in the second case AVENUE is the street type. The actions required in both cases are different.

Consider a case such as:

123 EAST WEST HIGHWAY and

123 WEST SOUTH STREET

In the first case, the name of the street is EAST WEST, and in the second case the name of the street is SOUTH. They both match to a pattern of * D | D | T, but each requires different actions.

Providing conditional values in patterns can correctly process such problem cases.

Simple conditional values

A simple condition is expressed by the usual pattern operand followed by an equal sign and a value. Alphabetic values must be in double quotes. A condition to distinguish ST. CHARLES from other street types would be *T = "ST" | +. This is a street type whose value is ST followed by an unknown alphabetic. This works because the street type of ST would follow an unknown, whereas ST for SAINT would precede an unknown. The set of patterns and actions to handle ST would be:

```
*T = "ST" | +
```

```
RETYPE [1] ? "SAINT"
```

The RETYPE action uses the ? and not the +, since + is not a fundamental token type. Rather, it is a pattern class with its type equal to ?. If a + is used as a RETYPE argument, it is, in any event, internally replaced by ?.

The value to be compared is based on both the standard abbreviation (if one exists) and the complete word. This prevents you from having to code all possible values, since ST and STREET would both match to the standardized abbreviation ST.

Suppose the word SOUTH was in the classification table. In this case, you could test explicitly for SOUTH by coding:

```
D = "SOUTH"
```

You could also test for any direction with the standard abbreviation S by coding:

```
D = "S"
```

for the operand.

Numeric values are coded without quotes. For example, * ^ = 1000 | ? would match to 1000 MAIN but not to 1001 MAIN.

The equality operator (=) tests both the standardized abbreviation and the original token value for equality to the operand. Two additional operators are available for testing equality to the abbreviation only or the original token value only. These are =A= and =T=. =A= tests the abbreviation only, while =T= only tests the original token value.

For example, you wouldn't want to recode ST. CHARLES as STREET CHARLES, which also has a standard abbreviation of ST, so it is preferable to test for equality with the original token value:

```
*T =T= "ST" | +
```

```
RETYPE [1] ? "SAINT"
```

This makes sure that ST was actually coded and not another value that maps to the same abbreviation. Similarly, use =A= if a test on only the abbreviation is desired. For example, you could use =A= to test:

```
D =A= "E"
```

In this case, it doesn't matter if the original word was EAST or E.

Series of conditional values

A series of conditional values is specified by delimiting the entries either by spaces or commas. For example:

T="ST", "AV", "PL" or

T="ST" "AV" "PL"

are equivalent and mean a street type whose standardized abbreviation is ST, AV, or PL. Numeric series can be represented in the same manner, except without quotation marks.

Again, the abbreviation equality operator =A= or the original value operator =T= may be used:

T=A="ST", "AV", "PL"

A series of values can be tested against the match key in a similar fashion:

^|T|{SN}="MAIN", "ELM", "COLLEGE"

In the case where a match key value is tested (instead of a normal pattern operand), the test must follow all pattern operands (including end-of-field). For example, the following is not valid since pattern operands follow the match key test:

^|{SN}="MAIN", "ELM", "COLLEGE"|T

Tables of conditional values

If it is necessary to specify a large number of conditional values, then tables of values may be used. Tables can be specified as follows:

@<table file name>

As an example, suppose you wanted to test a number to see if it is one of a series of postal codes. You can first prepare an ASCII

file with one line for each post code. As an illustration, assume this file is named postcode.dat and looks as follows:

90016

90034

90072

90043

...

A pattern matching city, state, and ZIP might look like:

?|S|^=@postcode.dat

This matches to cases such as "LOS ANGELES CA nnnnn". If the numeric operand is in the list, then the pattern matches, otherwise it does not. "LOS ANGELES CA 90016" would match, but "CHICAGO IL 12345" would not since the ZIP Code is not in the table.

The table filename may contain complete or relative path information.

Match keys can also be tested against tables of values:

^|T|{SN}=@strtname.dat

Since match key contents are not pattern operands, the test against a table of values must follow all pattern operands, including an end-of-field operand. For example:

^|{SN}=@strtname.dat|T

is not valid, since a pattern operand follows the table test.

Note that if values in the table are character strings, they need to be in uppercase.

Conditional expressions

If simple values or tables of values are not sufficient to qualify pattern matching, then conditional expressions may be coded. These have the following format:

```
<operand> [ <conditional expression > ]
```

The conditional expression is enclosed in braces immediately following the pattern operand. A simple conditional expression consists of an operand, a relational operator, and a second operand. The following are the relational operators:

- < original token is less than
- > original token is greater than
- = abbreviation or original token is equal to
- =A= abbreviation is equal to
- =T= original token is equal to
- <= original token is less than or equal to
- >= original token is greater than or equal to
- != abbreviation and original token are not equal to
- !=A= abbreviation is not equal to
- !=T= original token is not equal to

The operands can be any of the following:

- a variable name
- the match key contents for the current operand
- the match key contents for any field
- a literal
- a constant
- the special variables LEN and PICT

These concepts will be described in the following sections, along with examples.

Referencing current operand contents

The special operand `{}` indicates the contents of the current operand. To illustrate this concept, consider the city, state, and ZIP Code example again. You want the pattern to match only if a city and state are present and the ZIP Code is greater than 50000:

```
?|S|^[{ }>50000]
```

This pattern would match on:

```
CHICAGO IL 50104
```

but would not match on:

```
ALBANY NY 12345
```

To review, the pattern operands have the following meaning:

- Operand 1: One or more unknown words
- Operand 2: A class S word
- Operand 3: A numeric value greater than 50000.

When character literals are in an equality test, the standard abbreviation is tested, if one is available. If this fails the test, the original input is tested. For example:

```
T [ {} = "ST" ]
```

compares the abbreviation of the current operand (ST) to the literal ST.

```
T [ {} = "STREET" ]
```

compares the entire input operand against the literal (since it fails to compare to the abbreviation).

```
T [ {} =A= "ST" ]
```

compares only the abbreviation of the current operand to ST.

```
T [ {} =T="STREET"]
```

compares the original value of the token to STREET and not the abbreviation.

When comparisons (other than the equality operators) are specified:

```
T [ {} <="ST" ]
```

the original input is used rather than the abbreviation. This is true for any comparison to a literal value. In this case, if the original value were ST the result would be true, but if the original value were STREET then the result would be false.

Referencing match key contents

It is sometimes important to test a value placed in the match key from an earlier process or from a pattern–action pair that was previously executed. This can be accomplished by specifying the match key field name enclosed in braces.

For example, suppose you knew that two streets were named EAST WEST HIGHWAY. In ZIP Codes 20100 to 20300 the name of the street is EAST WEST, and in ZIP Codes 80000 to 90000, the name of the street was WEST and EAST is the direction. Suppose that the ZIP Code field in the match key was filled in by an earlier process and named {ZP}. The following pattern–action pairs could be used:

```
^|D="E"|D="W"|T="HWY"|[ {ZP} >= 20100 & {ZP} <= 20300 ]
```

```
COPY [1] {HN} ; move house number to {HN} field
```

```
COPY [2] temp ; concat EAST WEST and move to street name
```

```
CONCAT [3] temp
```

```
COPY temp {SN}
```

```
COPY_A [4] {ST} ; move HWY to street type field
```

```
^|D="E"|D="W"|T="HWY"|[ {ZP} >= 80000 & {ZP} <= 90000 ]
```

```
COPY [1] {HN} ; move house number to {HN} field
```

```
COPY_A [2] {PD} ; move EAST to direction
```

```
COPY [3] {SN} ; move WEST to street name
```

```
COPY_A [4] {ST} ; move HWY to street type field
```

This is the most complex pattern yet presented. The last operand states:

{ZP} the value in the ZIP Code match key field

>= is greater than or equal to

20100 the numeric value 20100

& and

{ZP} the value in the ZIP Code match key field

<= is less than or equal to

20300 the numeric value 20300

The logical operator & has not been discussed yet, but it is used to connect two conditional expressions. Notice that the condition is placed in a separate operand. It could also be placed in the operand for HWY as follows:

```
^|D="E"|D="W"|T="HWY" [ {ZP} >= 80000 & {ZP} <= 90000 ]
```

These two forms are identical. It is however, easier to read if the conditions are placed in separate pattern operands, as first shown.

When conditions only reference match key contents (and not any pattern operand), as in the above example with {ZP}, then the condition must follow all pattern operands. For example:

```
^[{ZP} = 80000]T
```

is invalid since the second operand does not reference an input field, and a third operand (T) follows the condition. This should be replaced with:

```
^[T][{ZP} = 80000
```

Referencing variables, literals, and constants

Numeric constants are referenced by coding a number. For example:

```
^[{ } = 10000]
```

This pattern operand matches on a number equal to 10,000.

Negative numbers and decimal points are not permitted in numeric constants.

Literals are character constants. They are represented by enclosing a string in quotes:

```
?[ { } = "MAIN" ]
```

This matches to the text MAIN.

If an unknown operand (?) is specified, then multiple words are concatenated to a single word. To match to CHERRY HILL, code

```
?[ { } = "CHERRYHILL" ]
```

A null or empty value is indicated by two consecutive quote marks.

```
?[ { } = "" ]
```

The user may use any of the relational operators for character strings. For example:

```
?[ { } > "MA" ]
```

will match on all strings starting with MA or greater. This includes MA, MAIN, NAN, PAT, but not ADAMS, M, LZ, and so on.

Remember that equality (=) is tested on the abbreviation for the operand first, if one exists, then on the full original operand. The other relational operators test on the full operand and not the abbreviation.

The user may define variables. Variables can be given any name desired. Variables are set to specific values by means of the actions. Variables can be tested for specific values within conditions. Variables must be named according to the following conventions: the first letter must be alphabetic, and the name may not exceed 32 characters. After the first character, any combination of alphas, digits, or underline characters may be used.

For example, suppose you set the variable postcode to the ZIP Code. You can test to see if the post code is 12345 as follows:

```
[postcode = 12345]
```

This type of condition can be a separate pattern operand or combined with a standard class. For example, the patterns `^[postcode = 12345]` and `^[postcode = 12345]` produce identical results.

If a user variable is set to a numeric value, its type is numeric. If it is set to a literal value, its type is character.

If a condition only references variable or match key fields and not current input operands, the condition must follow all operands. For example:

```
^[ [postcode = 12345] ] T
```

is invalid since the second operand does not reference an input field and a third operand follows. It should be replaced with:

```
^|T|[postcode = 12345]
```

Referencing the length of an operand

A special LEN qualifier is available. This represents the length of an operand. The expression must be coded as:

{ } LEN	Length of current operand
<variable> LEN	Length of contents of variable
{<match-key-field>} LEN	Length of match key field

For example, suppose you want to search for a nine-digit ZIP of the form 12345-6789. You want to be sure these are five- and four-digit numbers, respectively.

```
^[ { } LEN = 5 ] - | ^ [ { } LEN = 4 ]
```

If the numerics do not match the length, the pattern will not match.

Similarly to test the length of a variable, use the expression:

```
? [ temp LEN = 5 ]
```

This is true if the variable contains five characters.

Finally, to test the length of a match key, code the two-character field name within braces:

```
[ {SN} LEN = 20 ]
```

Referencing a template of an operand

There are sometimes cases where special formats must be tested. The PICT (picture) qualifier is available to accomplish this. Consider the case of Canadian postal codes. These are of the form character-number-character (space) number-character-

number. Examples include: K1A 3H4. The PICT qualifier is used to represent these sequences.

```
@ [ { } PICT = "cnc" ] | @ [ { } PICT = "ncn" ]
```

The @ matches to a complex type: mixed numerics and alpha. The picture defines how the numerics and alphas are distributed. cnc means character-number-character, and "ncn" means number-character-number.

Next consider British postal codes of the form character-character-number (space) number-character-character, such as AB3 5NW.

```
< [ { } PICT = "ccn" ] | > [ { } PICT = "ncc" ]
```

The PICT clause will work for match key values:

```
[ {ZP} PICT = "ccn" ]
```

or variable variables

```
[ temp PICT = "ncncn" ]
```

Only the equality (=) and inequality (!=) operators may be used with PICT comparisons.

Referencing a substring of an operand

A special form is provided in the patterns to test a portion of an operand. These portions are called substrings. The following forms are valid:

{ } (beg:end)	Substring of current operand
<variable> (beg:end)	Substring of contents of variable
{<match-key-field>} (beg:end)	Substring of match key field

The (beg:end) specifies the beginning and ending character to extract. The first character in the string is 1, the last character is -1, and so on.

For example, German style addresses have the street type indicated in the same word as the street name. Thus, HESSESTRASSE means HESSE STREET. The substring form can be used to test for these suffixes:

Consider an input address such as:

HESSESTRASSE 15

The pattern:

+ [{} (-7:-1) = "STRASSE"] | ^

would match to all words ending in STRASSE that were followed by a numeric.

Similarly variables and match key fields can be tested:

[temp(2:4) = "bcd"]

[{SN}(1:4) = "FORT"]

When conducting substring tests on multitoken values (?), remember that separating spaces are removed. Thus, to test MARTIN LUTHER KING specify "MARTINLUTHERKING".

Arithmetic expressions

Arithmetic expressions can be included as the left operand of a conditional expression. The following arithmetic operators are available:

- + addition
- subtraction
- * multiplication
- / division
- % modulus

Arithmetic is limited to one operation per expression. No parentheses are permitted. The modulus operation is the

remainder of an integer division. For example, $x \% 2$ is 0 if the number is divisible by two. It is 1 if the number is odd. At this point in the discussion, the complete format of a conditional expression can be presented:

<left-operand> <relational-operator> <right-operand>

The left operand may be:

variable name

{}

{ } PICT

{ } LEN

{<match key name>} PICT

{<match key name>} LEN

variable name PICT

variable name LEN

{<match key name>}

<arithmetic-expression>

The relational operators are <, >, <=, >=, !=, =

The right operand may be:

variable name

literal

constant

An arithmetic expression is

<left-arith-operand> <arith-operator> <right-arith-operand>

A left-arith-operand is a

variable name

{ <match key name>}

}

An arith operator is + - * / %

A right-arith-operand may be

variable name

constant

Examples of arithmetic expressions are

temp - 2 contents of temp - 2

{ } % 2 current operand value modulo 2

Consider a conditional expression for matching to even-numbered houses.

$^[\{\} \% 2 = 0]$

Even numbers are divisible by 2, thus the house number module 2 is zero.

Notice that the arithmetic appears on the left side of the relational operator (the equal sign).

A conditional expression is used to see if the current operand divided by three is greater than the contents of variable temp:

$^[\{\} / 3 > \text{temp}]$

Again note that the match key references and the arithmetic are to the left of the relational operator. Other examples of conditional expressions involving arithmetic are:

[temp * temp2 > temp3]

[{ZP} + 4 > 12345]

Combining conditional expressions

The preceding sections showed single conditional expressions for each pattern operand. Conditional expressions may be combined by using the logical operators:

& and

| or

To test for even-numbered houses greater than one thousand:

$^[\{\} \% 2 = 0 \& \{\} > 1000]$

To test for houses in the range of 1,000 to 10,000:

$^[\{\} >= 1000 \& \{\} <= 10000]$

To test for houses less than 100 or greater than 1,000:

$^[\{\} < 100 | \{\} > 1000]$

To test for even-numbered houses and the value in temp divided by 2 greater than 50, with ZIP Code greater than 12345:

$^[\{\} \% 2 = 0 \& \text{temp} / 2 > 50 \& \{\text{ZP}\} > 12345]$

Notice that no parentheses are permitted. All operations are executed left to right. Within a single bracket-delimited condition, AND and OR operators cannot be mixed. An error message is printed if such a case is encountered. Operator precedence can be obtained by using separate pattern operands if possible. For example:

$((a | b) \& (c | d))$ can be represented by:

$[a | b] | [c | d]$

The vertical lines within the brackets are logical OR operations, and those outside the brackets are operand separators. In this example, a, b, c, and d represent conditional expressions.

Actions

Whenever a pattern matches, the series of associated actions are performed. The actions were discussed informally in the previous sections. This chapter presents a systematic discussion of the actions.

Remember that each operand of a pattern (separated by the vertical line) has an operand number. For example, in the pattern:

```
^|?|T="ST"
```

there are three operands: [1], [2], and [3].

Copying information (COPY)

The COPY command is used to copy information from a source to a target. Its format is:

```
COPY <source> <target>
```

The <source> can be any of the following:

operand	a pattern operand ([1], [2], ...)
substring operand	a substring of the pattern operand
mixed operand	leading numeric or character subset
user variable	a user-defined variable
match key field	a key reference ({SN}, ...)
formatted field	a process reference ({CITY}, {STATE}, ...)
literal	a string literal in quotes ("SAINT")
constant	a numeric value

The <target> may be:

- match key field
- user variable

First, consider several simple examples. Consider a standard address pattern that would match to "123 N MAIN ST". This would be:

```
^|D|?|T
```

The number (operand [1]) should be moved to the house number {HN} field of the match key; the class D operand (direction) should be moved to the {PD} (prefix direction) field of the match key; the unknown operand (street name) should be moved to the {SN} field and the street type (class T) should be moved to the {ST} field. This is accomplished by the following pattern–action set:

```
^|D|?|T
COPY [1] {HN}
COPY_A [2] {PD}
COPY [3] {SN}
COPY_A [4] {ST}
EXIT
```

The COPY_A command is similar to the COPY command, except that the standardized abbreviation is copied to the match key rather than the raw value itself. A single COPY from a ? operand (street name in this case) will copy all words of the street name to the match key field. The words will all be concatenated together. Similarly, a single COPY from a ** operand will copy all tokens within the range of the **. They will be copied with no intervening spaces.

The following subsections will discuss the various options available with the COPY command.

Copying substrings

The simplest form of the COPY command is copying an operand to a match key value:

```
COPY [2] {SN}
```

A substring of an operand can be copied by using the substring operand form. This form is:

```
COPY <source>(b:e) <target>
```

where b is the beginning column of the string and e is the ending column.

For example:

```
COPY [2](1:1) {SN}
```

copies the first character (1:1) of operand 2 to the street name match key field.

```
COPY temp(2:4) {SN}
```

copies the second through fourth characters of the contents of the variable temp to the {SN} field.

-1 can be used to indicate the last character, -2 the next to last character, and so on:

```
COPY [2](-3:-1) {SN}
```

copies the last three characters of operand 2 to the street name field.

The substring operand only operates on standard operands or user variables.

Copying leading/trailing characters

There are four possible mixed operand specifiers. These are:

- (n) all leading numeric characters
- (-n) all trailing numeric characters

(c) all leading alphabetic characters

(-c) all trailing alphabetic characters

These specifiers can be used for standard operands or for user variables.

Consider an address such as 123A MAIN ST. You want the 123 to be recognized as the house number and the A to be recognized as a house number suffix. This can be accomplished as follows:

```
>|?|T
```

```
COPY [1](n) {HN}
```

```
COPY [1](-c) {HS}
```

```
COPY [2] {SN}
```

```
COPY_A [3] {ST}
```

```
EXIT
```

Notice that the first operand (>) is appropriate for token (leading numeric). These leading and trailing specifiers would almost always be used with > or < operands.

Copying user variables

A user variable may be the target and/or the source of a COPY. The type of a target user variable is determined by the type of the source. For example:

```
COPY [1] temp
```

```
COPY "SAINT" temp
```

```
COPY temp1 temp2
```

```
COPY temp1(1:3) temp2
```

The first line copies operand 1 to a variable named temp. The second line copies the literal "SAINT" to the variable temp. The

third line copies the contents of variable temp1 to temp2. The fourth line copies the first three characters of temp1 to temp2.

User variables may consist of 1–32 characters where the first character is alphabetic and the other characters are alphas, numbers, or the underscore character.

User variables are most often used with the CONCAT action to form one field. For example, if you knew that in EAST WEST HWY, the name of the street was EAST WEST, then the following patterns and actions would accomplish this:

```
^|D="E"|D="W"|T
COPY [1] {HN}
COPY [2] temp
CONCAT [3] temp
COPY temp {SN}
EXIT
```

The pattern tests for the specific directions E and W. After the house number is copied, the second operand is copied to the user variable temp. The contents of temp are now EAST. Notice the abbreviation is not copied, but the original string is. COPY copies the original operand contents, while COPY_A copies the abbreviation. The next line concatenates the second direction WEST to the variable temp. The variable now contains EASTWEST. Finally the contents of temp are copied to the street name field.

Copying match key fields and formatted fields

Match key fields can be copied to other match key fields:

```
COPY {HN} {HC}
```

A formatted field can also be copied to a match key field. For example, suppose there are three processes named CITY, STATE,

and POSTALCODE. The entire contents of the defined fields can be moved directly into the match key fields. For example, the user file specifies:

```
STANDARDIZE CITY (1–20)
STANDARDIZE STATE (21–41)
STANDARDIZE POSTALCODE (41–51)
```

COPY commands can be used to copy the raw data in these fields to the match key fields:

```
COPY {CITY} {CT}
COPY {STATE} {ST}
COPY {POSTALCODE} {ZP}
```

Copying standardized abbreviations (COPY_A)

The COPY command copies the raw operand to a target. The COPY_A action copies the standardized abbreviation for an operand to a target. Standardized abbreviations are coded for entries in the classification table for the process. They are not available for the hardwired classes, such as numeric, alpha unknown, and so on.

The COPY_A action can be used to copy the abbreviation of an operand to either the match key or a user variable:

```
^|?|T
COPY [1] {HN}
COPY [2] {SN}
COPY_A [3] {ST}
```

The third line copies the abbreviation of operand three to the street type match key field. Similarly,

```
COPY_A [3] temp
```

copies the standard abbreviation of operand three to the variable named temp. Abbreviations are limited to a maximum of 25 characters.

COPY_A may include a substring range, in which case the substring refers to the standard abbreviation and not the original token, as in COPY.

Copying with spaces (COPY_S)

Whenever an alpha operand (?) or a range of tokens (**) is copied to a match key or a user variable, the individual words are concatenated together. It is often the case that the match key would be used for generating mailing labels, and consequently, spaces should be preserved between words. This can be accomplished by using the COPY_S action instead of the COPY action. COPY_S requires an operand as the source and either a match key field or a user variable as a target.

Consider an input string such as: 123 OLD CHERRY HILL RD.

A standard copy would produce “OLDCHERRYHILL”, but COPY_S would operate as follows:

```
^|?|T
COPY [1] {HN}
COPY_S [2] {SN}
COPY_A [3] {ST}
```

The {SN} field of the match key would contain: OLD CHERRY HILL.

If the universal matching operand were used, then all tokens in the specified range would be copied. For example, consider removing parenthetical comments to a match key field named {PR}. For an input address such as:

```
123 Main St (Corner of 5th St) Apartment 6
```

the pattern:

```
\(|**|\)
COPY_S [2] {PR}
COPY_S [2] temp
```

would move “Corner of 5th St” to the match key field {PR}. The second action moves the same information to the user variable temp.

Copying the closest token (COPY_C)

When matching under uncertainty to entries in the classification table, one may want to copy the complete token spelled correctly, rather than copying an abbreviation. This is accomplished through the COPY_C action.

For example, consider a state name table with an entry such as:

```
MASSACHUSETTS MA S 800.0
```

then if Massachusetts were misspelled on an input record (for example, Massachusetts), you may want to copy the correct spelling to the match key:

```
COPY_C [operand number] {match-key-field}
```

will place the full correctly spelled token **MASSACHUSETTS** in the proper match key field.

Moving information (MOVE)

The MOVE action is similar to COPY except that the source is erased (made null). MOVE can be used to move either a user variable or a match key field to a match key field.

```
MOVE {HN} {HS}
MOVE temp {SN}
```

Concatenating information (CONCAT, PREFIX)

CONCAT is used to concatenate information to a user variable or a match key field. The source may be an operand, literal or user variable. For example, fractional addresses such as 1/2 and 1/4 can be copied to a single field in the match key as follows:

```
^|/|^  
COPY [1] temp  
CONCAT [2] temp  
CONCAT [3] temp  
COPY temp {HS}
```

The {HS} field will contain the entire fraction (1/2).

Suppose you wanted to copy two directions with spaces (for example, E W) into a single match key field:

```
D|D  
COPY_A [1] temp  
CONCAT “ “ temp  
CONCAT_A [2] temp  
COPY temp {PD}
```

Notice that a literal with a single space is concatenated to the variable. The same results could be obtained by concatenating directly into the match key field:

```
D|D  
COPY_A [1] {PD}  
CONCAT “ “ {PD}  
CONCAT_A [2] {PD}
```

The **CONCAT_A** action is identical to **CONCAT** except that the standard abbreviation is concatenated instead of the raw data.

The **PREFIX** action is available to prefix a string with data. **CONCAT** adds to the end of a string and **PREFIX** adds to the beginning of a string. The source for a **PREFIX** action may be an operand, a literal, or a user variable. The target may be a user variable or a match key field.

```
COPY “CHARLES” temp  
PREFIX “SAINT” temp
```

The variable temp contains SAINTCHARLES.

The **PREFIX_A** action is used to prefix the standard abbreviation instead of the raw data. The source must be an operand.

CONCAT and **PREFIX** permit substring ranges:

```
CONCAT [1](3:-2) {SN}
```

Columns 3 to the second to last column of the first operand will be concatenated to the street name field of the match key.

CONCAT_A and **PREFIX_A** also allow substring ranges; however, the substring refers to the standard abbreviation and not the original token.

Converting information (CONVERT)

The **CONVERT** action is used to convert data according to a lookup table or a user-supplied literal.

Introduction (converting place codes)

To illustrate the concept of conversion, suppose the input records contained numeric codes for places and you wanted to convert the numbers to actual place names. The user must first create a table file with two columns. The first column is the input value, and the second column is the replacement value. For this example, you will name the file “codes.tbl”, which will contain:

```
001 “SILVER SPRING”
```

002 BURTONSVILLE 800.0

003 LAUREL

....

Notice that multiple words are enclosed in double quotes. Suppose the first token in the input field contained the code. Just like the classification table, optional weights may follow the second operand to indicate that uncertainty comparisons may be used. In the above example, the string comparison routine would be used on BURTONSVILLE and any score of 800 or greater is acceptable. The following pattern would convert tokens according to the above table:

&

```
CONVERT [1] @codes.tbl TKN
```

The tokens remain converted for all patterns that follow. Thus, it will be as if the code was permanently changed to the text.

Retyping tokens

An optional token class may follow the TKN. The token will be retyped to this class if the conversion is successful. If no class is specified, then the token retains its original class.

For example:

+

```
CONVERT [1] @cities.tbl TKN C
```

```
CONVERT [1] @towns.tbl TKN T
```

```
CONVERT [1] @unincorp.tbl TKN U
```

would convert a single unknown alpha token based on the entries in the three files. If there was a successful conversion, the token would be retyped to either C, T, or U.

CONVERT considerations

This section describes some of the rules, caveats, and considerations to be taken into account when using the CONVERT action.

The source of a CONVERT can be an operand, a match key field, or a user variable. For example:

```
CONVERT temp @codes.tbl
```

```
CONVERT {CT} @codes.tbl
```

are both valid.

Entire path names may be coded for the convert table file specification:

```
CONVERT {CT} @..\values\valfile.dat
```

If the source of a CONVERT is an operand, then a third command argument is required:

```
CONVERT <operand> <table> TKN
```

```
CONVERT <operand> <table> TEMP
```

TKN means that the token table should be changed permanently. In other words, the tokens would be changed for pattern–action sets further down the current pattern file or for another process if that process used the same token table. If TEMP is coded, the conversion only applies to this set of actions. If it were desired to apply the conversion both to actions further down the program and the current set of actions, then two CONVERT actions should be specified (one using TKN for other action sets and processes, and one using TEMP for other actions in the current action set).

Converting multitoken operands

When converting multitoken operands, which were created by patterns using ** or ?, the format of the convert table depends on whether the third argument to CONVERT is TKN or TEMP. If the

third argument is TKN, then each token is separately converted. Thus, to convert SOLANO BEACH to MALIBU SHORES, the convert table must have the following two lines:

solano	malibu
beach	shores

This could produce unwanted side effects since any occurrence of SOLANO would be converted to MALIBU and any occurrence of BEACH would be converted to SHORES.

To avoid this situation, the TEMP option of the CONVERT must be used. In this case, the combined tokens are treated as a single string with no spaces. Thus, SOLANOBEACH becomes the representation for a ? pattern containing the tokens SOLANO and BEACH. The following entry in the CONVERT table will accomplish the proper change:

solanobeach	“malibu shores”
-------------	-----------------

In this convert table there must be no spaces separating the concatenated tokens. When copying the converted value to a match key, COPY and COPY_SP will produce the same result.

Assigning fixed values

CONVERT may also be used to assign a fixed value to an operand or match key. This is accomplished by

```
CONVERT <operand> <literal> TEMP | TKN
```

```
CONVERT <match-key-field> <literal>
```

For example to assign a city name of LOS ANGELES to a match key, either the COPY could be used or:

```
CONVERT {CT} “LOS ANGELES”
```

More importantly, it can be used to convert an operand to a fixed value:

```
CONVERT[1]“LOS ANGELES”TKN
```

TKN makes the change permanent for all actions involving this record, and TEMP makes it temporary for this set of actions.

An optional class may follow the TKN. Since converting to a literal value is always successful, the retyping will always take place.

Converting prefixes and suffixes (CONVERT_P, CONVERT_S)

When a single token may be composed of two distinct entities, a CONVERT-type action may be used to separate and standardize both parts. Such tokens may occur in German addresses where the suffix STRASSE may be concatenated onto the street's proper name, for example, HESSESTRASSE. If a list of American addresses has a significant error rate, one may need to check for occurrences of dropped spaces such as in MAINSTREET. To handle cases such as these, the CONVERT_P action may be used to examine the token for a prefix and CONVERT_S for a suffix.

Syntax

The syntax for CONVERT_P and CONVERT_S is almost the same as for CONVERT. The first difference is that CONVERT_P and CONVERT_S must use a convert table. The second difference is that an optional fifth argument may be specified:

```
CONVERT_P <source> @table_name TKN/TEMP <retype1>  
<retype2>
```

```
CONVERT_S <source> @table_name TKN/TEMP <retype1>  
<retype2>
```

<source> can either be an operand, a match key field, or user variable.

In the optional fourth argument, <retype1> refers to the token class that will be assigned to the prefix (CONVERT_P) or suffix (CONVERT_S) part of the token.

In the optional fifth argument, <retype2> refers to the token class that will be assigned to the body (or root) of the token.

Temporary conversion

Similar to the CONVERT action, there are two modes that can be specified: TEMP or TKN. You will first consider the TEMP mode, where the conversion applies to this set of actions only.

```
CONVERT_S [1] @suffix.tbl TEMP
```

would convert the suffix of the first operand according to the entries in the table suffix.tbl.

Consider an operand value of

```
HESSESTRASSE
```

with a table entry in suffix.tbl such as:

```
STRASSE STRASSE 800.0
```

Operand [1] would be replaced with the value:

```
HESSE STRASSE
```

Notice that there is a space now between the words. Subsequent actions in this pattern–action set would operate as expected. For example, COPY_S, CONCAT_S, and PREFIX_S would all copy the two words HESSE STRASSE to the target. COPY, CONCAT, and PREFIX would copy the string without spaces. For example, if the table entry reads:

```
STRASSE STR 800.0
```

then HESSE STR would be the result of the convert. COPY_S would preserve both words, but COPY would copy HESSESTR as one word.

The CONVERT argument can be an operand (as in the example), a match key field, or a user variable, with equivalent results.

Permanent conversion

When operating on an operand with mode of TKN, one would generally specify a retype argument, which would apply to the suffix (or prefix for CONVERT_P). Consider the following CONVERT_S statement:

```
CONVERT_S [1] @suffix.tbl TKN T
```

with an operand value of

```
HESSESTRASSE
```

and a table entry in suffix.tbl such as:

```
STRASSE STRASSE 800.0
```

HESSE would retain its class of ? (since no fifth argument is specified to retype the body or root of the word) and STRASSE would be given the retype for the suffix, for example, T for street type. Further actions on these two tokens would require a pattern of:

```
?|T
```

If no retype class had been given, both tokens would retain the original class of ?.

One may wish to retype both the prefix/suffix and body. When checking for dropped spaces, a token such as APT234 may occur. In this case, the token would have been found with a class of < (leading alpha) and an optional fourth argument could retype the

prefix APT to M for multiunit, and an optional fifth argument could retype the body 234 to ^ for numeric. For example:

```
CONVERT_P[1]@prefix.tbl TKN M ^
```

prefix.tbl would contain an APT entry.

Obviously, if one wishes to retype just the body, a dummy fourth argument must be supplied, which repeats the original class.

Side effects

One must be careful about avoiding side effects since the ? class can match to more than one token. Assume suffix.tbl has the following lines:

```
STRASSE STRASSE 800.
```

```
AVENUE AVE 800.
```

The following pattern/actions:

```
^|?
```

```
COPY [1] {HN}
```

```
CONVERT_S [2] @suffix.tbl TEMP
```

```
COPY_S [2] {SN}
```

```
EXIT
```

with input of:

```
123 AAVENUE BB CSTRASSE
```

would result in:

```
123 AAVENUEBBC STRASSE
```

However, the pattern/actions of:

```
^|?
```

```
CONVERT_S [2] @suffix.tbl TKN T
```

```
^|+|T|+|+|T
```

```
COPY [1] {HN}
```

```
COPY [5] {SN}
```

```
COPY [6] {ST}
```

```
EXIT
```

with input of:

```
123 AAVENUE BB CSTRASSE
```

would result in:

```
123 C STRASSE
```

and:

```
COPY [1] {HN}
```

```
COPY [2] {SN}
```

```
COPY [3] {ST}
```

would result in:

```
123 A AVE
```

In other words, when grabbing/concatenating alphas with the pattern ?, CONVERT_P/_S operates on the entire concatenated string for user variables, match key fields, and operands [mode = TEMP]. For operands [mode = TKN], each token in the token table that comprises the operand is examined individually, and new tokens corresponding to the prefix/suffix are inserted into the table each time the prefix/suffix in question is found.

Retyping operands (RETYPE)

The RETYPE action is used to change the type of an operand in the token table, to optionally change its value, and to optionally change its abbreviation. The format of the retype operand is:

```
RETYPE <operand> <class> [<variable> | <literal> ]  
[<variable> | <literal> ]
```

A basic concept of writing standardization rules is the concept of filtering. Phrases and clauses can be detected, processed, and removed from the token table. Removing them can be accomplished by retyping them to a null class (0). Null classes will not match to any patterns.

For example, suppose you want to process apartments and remove them from the address. Removing them converts an address such as:

```
123 MAIN ST APT 56
```

to a standard form:

```
123 MAIN ST
```

without interference from the apartment number clause. Consider the following rule to accomplish this:

```
%M | &  
COPY_A [1] {UT}  
COPY [2] {UV}  
RETYPE [1] 0  
RETYPE [2] 0
```

An apartment followed by any single token is detected. The fields are moved to the match key and retyped to null, so they don't match in any future patterns.

A third operand is available to replace the text of a token. For example, if you wish to recognize streets such as ST. CHARLES and replace the ST with the word SAINT:

```
*! ? | T = "ST" | +  
RETYPE [1] ? "SAINT"
```

This set scans for a street type whose value is ST followed by a single alphabetic word. The RETYPE changes the street type

operand to an unknown alpha (?) and replaces the text with SAINT. Now a case such as

```
123 ST CHARLES ST
```

becomes

```
123 SAINT CHARLES ST
```

and this will match to the standard `^ | ? | T` pattern. Notice that the `?` is used for the retype class. This is important because you want the SAINT to be considered to be the same token as the neighboring unknown alpha tokens. Notice that the first operand of the pattern (`*!?`) ensures that the ST is not preceded by an unknown. This keeps input such as:

```
MAIN ST JUNK
```

from being standardized into:

```
MAIN SAINT JUNK
```

A fourth operand is available for the RETYPE command to change the standard abbreviation of the operand. If this operand is included, then a third argument must also be present to change its value.

For example,

```
ST 123
```

may mean SUITE 123.

Suppose the standard abbreviation for SUITE were STE. You would need to change the standard abbreviation as follows:

```
T=A="ST" | ^  
RETYPE [1] M "SUITE" "STE"
```

Now this will be interpreted in future patterns as SUITE 123, and it will be given the standard abbreviation STE.

For the optional third and fourth arguments, a substring range is permitted.

For example, with an input of

8 143rd St

and a pattern/action set of:

^|>|T

COPY [2](n)temp

RETYPE [2]^temp(1:2)

143 will be copied to the variable temp, 14 will replace the contents of the second token, and its class will become numeric (^).

RETYPE will reclassify all elements of a possibly concatenated alpha class (?) or universal class (**) if no subfield ranges (n:m) are specified and will reclassify only those tokens within the subfield range if a range is specified.

For example, with input of:

15 AA BB CC DD EE FF RD

The following patterns/actions will have the described effect:

^|?|T

RETYPE [2]0 ;Sets AA to FF to NULL class

^|3|T

RETYPE [2]0 ; Sets CC to NULL class

^(2:3)|T

RETYPE [2]0 ; Sets BB and CC to NULL class

^-2|T

RETYPE [2]0 ; Sets EE to NULL class

Retyping multiple tokens (RETYPE)

A special pattern–action set is available to retype multiple occurrences of a token. The pattern must have the format:

<num>*<class>

followed by a standard RETYPE statement referencing operand [1]. The number must be an integer from 0 to 255 and indicates the number of occurrences referenced. Zero means that all occurrences should be scanned.

For example, suppose you processed hyphenated house numbers such as 123-45 Main St., and you would now like to remove all hyphens from the tokens that remain. This can be accomplished by

0 * -

RETYPE [1]0

This scans for all hyphens and retypes them to null. Suppose you wanted to retype two numeric tokens to an unknown type with a value of UKN. This can be accomplished by:

2 * ^

RETYPE [1]?“UKN”

The entire pattern is restricted to this simple format when multiple occurrences are referenced.

SOUNDEX phonetic coding

The SOUNDEX action is used to compute a SOUNDEX code of a match key field and move the results to another match key field. SOUNDEX codes are phonetic keys that are useful for blocking records in a matching operation. SOUNDEX is an excellent blocking variable since it is not very discriminating and yet can

be used to partition the files into a reasonable number of subsets. The SOUNDEX action has the following format:

```
SOUNDEX <source-field> <target-field>
```

For example, the action:

```
SOUNDEX {SN} {XS}
```

computes the SOUNDEX of the street name field and places the four character result in the XS match key field.

The RSOUNDEX (reverse SOUNDEX) action is the same as the SOUNDEX action except that the phonetic code is generated from the last nonblank character of the field and proceeds to the first. This is useful for blocking on fields where the beginning characters might be in error.

The SOUNDEX and RSOUNDEX actions are generally used in the post action section, so they are executed after pattern matching is complete for the record.

Post actions must occur prior to any pattern–action sets and must be preceded by the line

```
\POST_START
```

and followed by the line

```
\POST_END
```

Terminating pattern matching (EXIT)

The EXIT action is used to quit the pattern matching program for this process in this record. This prevents further pattern–action pairs from being executed.

For example:

```
^|D|?|T
COPY [1] {HN}
COPY_A [2] {PD}
COPY [3] {SN}
COPY [4] {ST}
EXIT
```

No other processing is required after this pattern–action set.

Subroutines (CALL)

Subroutines are available to facilitate fast processing. Since pattern–action sets are executed sequentially, it is faster to test for a generic type and call a subroutine to process that type. This is best illustrated by an example:

```
%I M
CALLAPTS
```

If a multiunit type is detected anywhere in the input, the routine APTS will be called to process the apartment numbers. Subroutine names are formed according to the same rules as variables names (1–32 characters, the first being alphabetic).

Writing Subroutines (SUB, END_SUB)

A subroutine is delimited in a manner similar to the POST actions. Subroutines have a header and a trailer line:

```
\SUB <name>
——
——  subroutine body
——
\END_SUB
```

As many `\SUB \END_SUB` sets as required may be coded. Subroutines are invoked by a `CALL` action:

Consider this example of a rules file organization:

```
\POST_START
post actions
  \POST_END
  %1 M
  CALLAPTS
  %1 R
  CALLROUTES
```

Additional patterns and actions

```
\SUBAPTS
Apartment processing patterns and actions
  \END_SUB
  \SUBROUTES
```

Route processing patterns and actions

```
\END_SUB
```

Notice that all subroutines are coded at the end of the file. The order of the subroutines themselves is unimportant. The subroutines contain the standard pattern–action sets as would be found in the main rules.

It is important to note that `CALL` actions are not permitted from within subroutines. In other words, subroutine invocations must come from the main program.

Control is returned back to the main program either when the `\END_SUB` is reached or when a `RETURN` action is executed. All actions except the `CALL` action are permitted from within a subroutine.

Returning from a subroutine (RETURN)

The `RETURN` action is available to return control from a subroutine to the main program. A `RETURN` is not required immediately preceding the `\END_SUB` statement.

Summary

The following is a summary of the sources and targets allowed for all actions.

ACTION	SOURCE	TARGET
CONCAT_A	operand	user variable
	operand	match key field
PREFIX_A	operand	user variable
	operand	match key field
COPY_A	operand	user variable
	operand	match key field
CONCAT	operand	user variable
	operand	match key field
	literal	user variable
	literal	match key field
	user variable	match key field
	user variable	user variable
PREFIX	operand	user variable
	operand	match key field
	literal	user variable
	literal	match key field
	user variable	match key field
	user variable	user variable
MOVE	user variable	match key field
	match key field	match key field
COPY_S	operand	match key field
	operand	user variable

ACTION	SOURCE	TARGET
COPY_C	operand	match key field
	operand	user variable
COPY	operand	match key field
	operand	user variable
	formatted field	match key field
	match key field	match key field
	literal	match key field
	literal	user variable
	user variable	match key field
	user variable	user variable
CONVERT	match key field	
	user variable	
	operand	
SOUNDEX	match key field	match key field
	user variable	match key field
RSOUNDEX	match key field	match key field
	user variable	match key field

Developer's Kit software tools reference

C

IN THIS APPENDIX

- Developer's Kit software tools reference

This appendix is a reference for the Developer's Kit software tools.

Developer's Kit software tools reference

The Geocoding Developer's Kit tools consist of the following items:

STANDARDIZER EDITOR (STANEDIT.EXE)

Windows version of Interactive Standardizer.

STANEDIT is a Windows-based and internationalized interactive standardization program. STANEDIT is used for syntax checking and debugging of standardization pattern rules. STANEDIT is a 32-bit executable that runs under Windows 95/98 or Windows NT.

Additional Tools provided in STANEDIT:

ENCODPAT

Standardizer Pattern Rule Encryption Program

ENCODPAT encodes the standardizer pattern rules files for use with ArcView or MapObjects 2.0 or above. ArcView or MapObjects requires the pattern rules files to be encoded using ENCODPAT.

Unencoded pattern rule files and ASCII format geocoding files

The unencoded versions of pattern rules files used by ESRI geocoding products. By convention, the unencoded versions of .pat files have a .xat extension. The unencoded pattern rule files have been included with a complete set of geocoding standardization process and match rule files. To be used, the pattern rules files must be encoded and renamed to .pat.

Avenue tools and examples

A set of Avenue™ scripts used in the geocoding development process. The scripts include an example of Avenue script-based geocoding, bulk event standardization used for standardizer debugging, and mkstyles.ave used to create new AddressStyles.

Visual Basic sample application for MapObjects

A sample geocoding application for MapObjects 2.0 or above in Microsoft® Visual Basic®.

Sample data

Sample data is provided for use in demonstrations and geocoding development. The sample data consists of the street network from the ArcView sample data for Atlanta and a portion of ESRI's Redlands database. Both datasets are in shapefile format. Address event files for both datasets have also been included. The file cust.lst.dbf can be geocoded to atlanta.shp, and customer.dbf can be geocoded to redlands.shp.

Installation

The Geocoding Developer's Kit software can be installed by running setup.exe. All installed files will be stored in the specified GeoKit folder. No files will be installed in the systems folders. To uninstall the Kit, run Add/Remove Programs in the Control Panel.

Supported platforms

The GeoKit software is supported in Windows 95/98 and Windows NT only.

Contents

The Geocoding Developer's Kit software directory structure consists of the following subfolders of "geokit":

.GeoKit	Top-level GeoKit directory
.Bin	GeoKit binary executables and DLLs
.Geocode	Pattern rules and other geocode files
.Samples	
.AVScripts	ArcView Avenue scripts

.\Data	Sample data
.\Exercises	Sample answers for some exercises
.\Vb	Sample application for MapObjects in Visual Basic

The remaining sections of this appendix contain additional information on the use of STANEDIT, ENCODPAT, and the Avenue scripts. Additional information, such as the syntax for standardizer pattern rules, can be found elsewhere in this manual.

STANEDIT (Standardizer Editor)

Description

STANEDIT is a 32-bit Windows-based and internationalized interactive standardization program.

Usage

STANEDIT can be used for address standardization rules syntax checking and debugging. It lets you perform several functions, including opening an existing process, creating a new process, editing the geocode files, setting preferences for the display format and text editor, encoding the pattern file, and calculating the maximum composite score based on a Match Rule (.mat) file.

To open STANEDIT, you can either double-click the Program shortcut or choose the program in the Start menu. Click Open Process in the File menu to open an existing standardization process. By default, the standardization rule files are stored in the Geocode folders under the GeoKit installation folder. It can be modified in the Preferences menu.

Enter an address you want to standardize in the Standardizer window. The window is nonmodal. It can be displayed in a single-line form or multiline form as specified in the Preferences menu.

The patterns used for standardizing the input address will be displayed on the main window.

The output file specified by the OUTFILE keyword in the .stn file also contains the patterns that the input address was standardized against. You can examine the output file after the process is closed. The file will be overwritten every time the process is opened.

```
pattern matched: ^| ?| T| $ ; Standard form: 123 MAIN ST
HN=380
SN=NEW YORK
ST=ST
```

```
380 NEW YORK ST   380 new york street
```

Files associated with the selected process can be edited by choosing the options in the Edit menu. Some optional tables may be used with the pattern rules. You can choose Table (TBL) under the Edit menu and specify a filename that you want to edit, or choose New Table in the File menu to create a new table. In addition, you can specify if you want to use a text editor other than Notepad in the Preferences menu. After editing and saving the files, choose Reload under the Edit menu to reload the process.

Click New Process in the File menu to create a new standardization process, and the files will be opened with a text editor. Edit the files and add the rules accordingly.

When you are finished debugging the rule files, choose Encode PAT under the Tools menu to encode the rule file.

STANEDIT also performs pattern rules syntax checking. If STANEDIT detects syntax errors in the input standardization process, a set of diagnostic error messages is displayed. STANEDIT will close the process after you click OK on the Error message box. You can choose the Last File option in the Edit

menu to open the problem file and correct the errors. Then choose Open Process in the File menu to open the process again.

STANEDIT outputs the standardized address in the order of the match key dictionary.

Multiple addresses can be entered, one after another.

ArcView and MapObjects may generate unpredictable results and errors if standardization process syntax errors exist or the *.pat file is not encoded.

Discussion

STANEDIT is a complete interactive address standardizer. By modifying the input standardization process, *.stn file, STANEDIT can be used for standardizer process file syntax checking and pattern rules debugging. The following steps are an example of how to use STANEDIT in the standardization debugging process.

1. Edit the standardization process *.stn file. Add the DEBUG and OUTFILE keywords. The DEBUG keyword puts the standardizer into debugging mode, and the OUTFILE keyword sets the debugging output to <file_name>.

```
RECORD256
```

```
TYPEASCII
```

```
INTERACTIVE
```

```
DEBUG          <-      Line added
```

```
STANDARDIZE us_addr
```

```
OUTFILE _us_addr.txt <-      Line added
```

The DEBUG and OUTFILE keywords should be removed prior to using the standardization process in ArcView.

2. Run STANEDIT and load a standardization process. Follow this by entering an address you want to standardize in the Standardizer menu. The standardized results will be displayed in a single line or individual text box next to a MatchKey named as defined in the process match key dictionary.
3. Examine the process of standardization in the output file specified by the OUTFILE keyword. In addition to displaying the standardization results on the STANEDIT window, the OUTFILE also contains the patterns which the input address was standardized against, that is, the exact process by which the address was standardized.

```
Pattern matched: ^|?|T|$ ; Standard form: 123 MAIN ST
```

```
HN=380
```

```
SN=NEW YORK
```

```
ST=ST
```

```
380 NEW YORK ST  380 new york street
```

If the output file specified with the OUTFILE command does not contain a pathname, it will be written to the same location as the standardization process command file. Otherwise, the file will be written to the pathname specified.

4. When STANEDIT loads, it performs compile-time syntax error checking on the standardization process files. Most syntax errors are displayed as error messages on initialization. Some errors, such as missing .tbl files, are not listed as error messages.

```
Error: invalid command in line
```

```
RETYE [3]0
```

```
line number = 14
```

```
file us_addr.pat
```


The example above shows a compile-time syntax error found at line 14 in the file `us_addr.pat`.

For additional documentation about STANEDIT and pattern rule files, see Appendix B.

ENCODPAT

Description

ENCODPAT is used for encrypting standardization pattern rule files for use in ArcView and MapObjects. The standardization pattern rule files are required to be encrypted using ENCODPAT.

STANEDIT reads and edits the nonencoded pattern rules in an ASCII file with the `.xat` file extension. ENCODPAT converts the file into a binary file with the `.pat` file extension. STANEDIT provides an option to encode the pattern rule file of the selected process. To encode the pattern rule file, click the Tools menu in STANEDIT and click Encode PAT.

If you prefer, you can run ENCODPAT.EXE in a DOS prompt without STANEDIT:

```
C:\geokit\geocode>..\bin\encodpat
```

```
encodpat <in_file> <out_file>
```

```
C:\geokit\geocode>..\bin\encodpat us_addr.xat us_addr.pat
```

Usage

In STANEDIT, choose Encode PAT under the Tools menu.

In DOS Prompt: ENCODPAT <in_file> <out_file>

Arguments

<in_file>: The name of the input standardization pattern rules file.

<out_file>: The name of the output encoded pattern rules file.

Notes

ENCODPAT is a one-way file encryption program and can't be used to unencode a file.

Also, standardization pattern rule files must be encoded using ENCODPAT to be used in ArcView or MapObjects.

Discussion

ENCODPAT is only required for ArcView or MapObjects geocoding. There is no need to deploy the nonencoded pattern rule files with your applications.

Avenue tools and examples

Description

The scripts directory contains the following Avenue tools and examples:

geocode.ave: Sample Avenue-based geocoding. This script is used in the Section 3 exercise.

geocode2.ave: Advanced sample Avenue-based geocoding.

stantest.ave: Batch standardizer test script. This script reads and standardizes address events from a Vtab and writes its results to a DOS text file.

mkstyles.ave: AddressStyle creation script. This script defines a set of AddressStyles and creates the style ODB file addstyle.db. GEOPUB30.dll must be loaded for this script.

us_addr.ave and other .ave files: key file creation script. This script creates a match key file for the Geocoding Editor used by ArcView only. The key filename must be the same as the standardization process name, such as us_addr.key, us_intsc.key.

For additional information, see the individual scripts. Also, remember that you may need to edit the scripts—for example, you may need to change the path or filenames.

MapObjects geocoding example

Description

A geocoding application for MapObjects 2.0 in Microsoft Visual Basic 6.0 is included. It demonstrates how to set up the geocoding objects with the rule files and run geocoding in MapObjects.

Sample data

Sample data is provided for use in demonstrations and geocoding development. The sample data consists of the street network from the ArcView sample data for Atlanta as well as a portion of ESRI's Redlands database. Both datasets are in shapefile format. Address event files for both datasets have also been included. The file cust_1st.dbf can be geocoded to atlanta.shp, and customer.dbf can be geocoded to redlands.shp.

Matching and standardization file conventions and limits

D

IN THIS APPENDIX

- **Conventions and limits**

This appendix contains the list of matching and standardization file conventions and limits that are imposed by ArcView, MapObjects, and STANEDIT. These conventions and limits should be adhered to when developing standardization pattern rules for use in ArcView or MapObjects.

Conventions and limits

Pattern file conventions

STANEDIT/ENCODEPAT requires that the first line of all pattern rule files must be a comment line.

STANEDIT/ENCODEPAT also requires that the last line contains a carriage return.

A blank line must follow all subroutine calls.

Default parsing parameters (STRIPLIST and SEPLIST)

The STRIPLIST and SEPLIST contain default rules that define what constitutes a token or operand as defined in the pattern file.

STRIPLIST “ ,\’;:”

SEPLIST “ ()-/#&;:”

Characters in the list must be enclosed in quotation marks. The quotation mark itself may not be in either list. If no list is coded for a standardization process, the default lists are used.

Limits

All values listed below are the maximum values allowed in ArcView and STANEDIT.

ArcView

Number of fields for an address style supported in the ArcView graphical user interface (GUI), that is, the theme’s geocoding properties dialog box: 24

Number of standardization processes (*.stn files) in one address style: 3

Length of an input address using the Locate dialog box and the aMatchkey.SetKey request: 1024

Matching specification (*.mat)

Number of VAR statements: 256

Size of field name in the VAR statement: 32

Total length of all fields in VAR statements: 255

Number of MATCH statements: 40

Pattern file (*.pat)

Number of value files, such as *.tbl and *.dat: 10

Number of tokens per record: 30

Number of user-defined variables used in a condition or action statement: 15

Number of conditionals within a single operand: 10

Length of user-defined variable name: 20

Length of contents in a user-defined variable: 40

Number of fixed and floating position qualifier and cut class qualifiers within a single pattern field (for example, %!T*! has three qualifiers): 4

Number of ranges within a single operand: 4

Number of characters or numeric values in a single operand: 10

Number of actions: No limit but patterns and actions per process must not exceed a 16K buffer

Classification table (*.cls)

Length of token abbreviation: 25

Glossary

actions

Commands in the pattern file that work in conjunction with tokenized addresses to do many things including filtering noise, converting an operand to other values based on a lookup table, copying a value or operand to a match key field or variable, or invoking a subroutine.

address

A description of a location, most often consisting of specific elements arranged into a particular format.

address data

Material containing the parameters of the location being searched in a geocoding search. The address data may consist of one individual address or a table containing many addresses.

address data format

The arrangement of location-specific information, most often consisting of such address elements as house number, street direction, street name, street type, city, and postal code.

address data model

A geodatabase designed specifically to accommodate address-related material, such as streets, zones, ranges, and so forth. An address data model facilitates address data storage.

address element

A part of an address. A house number, street direction, or street name are all examples of address elements.

address field

A column in a table, containing one or some address elements. Address fields can be present in reference data, address data, or both.

address format

An address format defines the structure of addresses and a method of matching that can be used for a specific application.

address locator

One of the entities in the geocoding framework that acts to combine the style-specific guidelines and reference data. The address locator defines the technique to be used by the geocoding engine in interpreting the address against the rule base files. The file containing information about a geocoding service is distinguished with a .loc file extension.

Address Locator Properties dialog box

The primary interface in ArcGIS used to create or modify the geocoding service. The geocoding service properties dialog box is accessible through both ArcCatalog™ and ArcMap™.

address matching

See geocode.

address standardizer

The address standardizer is what you enter input information into. The input information is found in the first column of the classification table.

alternate name

An additional name used to refer to an address feature. In the geocoding service, alternative names are used often for street names. When a street is known by more than one name, such as by a highway number and a street name, the alternative name can be included in the search to replace the related address element.

API (application programming interface)

See application programming interface.

application programming interface (API)

A defined and documented set of tools or functions that application developers use to build or customize a program or set of programs. APIs can be built for programming languages such as C, COM, Java, and so on.

arithmetic expressions

Expressions in the pattern file that can be <left-arith-operand>, <arith-operator>, or <right-arith-operand>.

blocking

Occurs during indexing to reduce the number of potential matches that need to be checked.

booster index

A geocoding index designed to reduce the time and resources needed for a geocoding search.

candidate

A location found on the reference data that has the possibility of being a match for the address being searched.

class

Used in the pattern file to specify the rules that will be used to interpret the elements of an address. A class must be a single character.

classification table

A standard ASCII file in the geocoding rule base that identifies and classifies keywords that may appear in an address, such as street types and directions. The file name is always <file>.cls.

classifying

The process of assigning a class to a part of an address to standardize it.

client-side address locator

An address locator created and used on the same computer.

command file

A file in the geocoding rule base that specifies the standardization commands and processes. The file name is always <file>.stn.

comparison threshold

Degree of uncertainty that can be tolerated in the spelling of the keyword, such as phonetic errors, random insertion, deletion, replacements, and the transposition of characters. The score is weighted by the length of the word, because small errors in long words are less serious than errors in short words.

composite weight

The sum of the individual weights for all field comparisons. The composite weight provides a reference for how good a match is.

CONCAT

A command in the pattern file that concatenates information to a user variable or a match key field. The source can be an operand, a literal, or a user variable.

CONCAT_A

A command in the pattern file that concatenates the standard abbreviation of the operand.

conditional expressions

Expressions in the pattern file that are enclosed in square brackets immediately following the operand. Conditional expressions can include <left-operand>, <relational-operator>, or <right-operand>.

conditional patterns

One type of pattern rule. Conditional pattern rules allow patterns to match only under specified circumstances.

conditional pattern values

Conditional values in patterns can correctly process problem cases, such as ST to SAINT. There are several types of conditional values, including simple conditional values, a series of conditional values, and tables of conditional values. See also simple conditional values, series of conditional values, table of conditional values.

constants

Numeric constants are referenced by coding a number. Negative numbers and decimal points are not permitted in numeric constants.

CONVERT

A command in the pattern file that converts data according to a lookup table instead of a user-supplied literal.

COPY

A command in the pattern file that copies the entire string (all words are concatenated). The form of the command is a <source> and a <target>.

COPY_A

A command in the pattern file that copies the standard abbreviation coded in the classification table for an operand to a target.

COPY_S

A command in the pattern file that preserves spaces between words when you copy them.

DEBUG

Keyword in the command file that puts the standardizer into debugging mode.

dynamic feature class

A feature class consisting of point features associated with the address elements in an address data table that will change based on any changes made to the address data table.

ENCODPAT

One of the tools that comes with STANEDIT in the Geocoding Developer Kit. ENCODPAT is a pattern rule encryption program that is used to encode standardization pattern rules files for use in ESRI's geocoding products. ENCODPAT converts an ASCII .xat file into a binary .pat file.

end of field

A specifier in the pattern file denoted by \$. This specifier matches to the end of the field instead of matching to any real token. It is used to ensure that no tokens are left in the field after the pattern.

end offset

A distance away from the endpoint of a street feature. The end offset is an adjustable value that dictates how far away from the end of a line feature an address location should be matched. Using a side offset will eliminate the point feature from being placed directly over or beyond intersecting line features.

EXIT

A command in the pattern file that quits the pattern matching program for this process in this record, which prevents further pattern/action pairs from being executed.

file

A concatenated method of referring to a shapefile.

fixed position specifier

A specifier in the pattern file denoted by %n. It specifies the position at a particular operand in the input string.

floating position specifier

A specifier in the pattern file denoted by *. Used to modify the positioning of the pattern matching. The class immediately following the specifier is a floating class. The pattern is searched until there is a match or the entire pattern is scanned.

GBF (Geographic Base Files)

See reference files.

GDB

Abbreviation of geodatabase. See geodatabase.

geocode

The process of assigning an x,y coordinate value to the description of a place by comparing the descriptive location-specific elements to those present in the reference data.

Geocoding Editor

Lists the match key fields for the candidate list (with the exception of any identifier that starts with X—for example, XS and XR).

geocoding engine

One of the entities in the geocoding framework that drives the geocoding process.

geocoding index

A table created in conjunction with a geocoding service, designed to decrease the resources necessary to perform a geocoding search. These tables contain a list attributed from the reference attribute table as well as a coded value for the attribute. When geocoding, the same index values are produced for your address query and are matched against the geocoding index.

geocoding platform

A conceptual entity of the geocoding framework that combines the interaction of the ArcGIS interface with the input parameters set in the geocoding service and the processes of the geocoding engine.

geocoding process

The steps involved in translating an address entry, searching for the address in the reference data, and delivering the best candidate(s). These steps include parsing the address, standardizing abbreviated values, assigning each address element to a category known as a match key, indexing the needed categories, searching the reference data, assigning a score to each potential candidate, filtering the list of candidates based on the minimum match score, and delivering the best match. The process requires reference files, input event records, and software.

geocoding style

A template on which a geocoding service is built. Each template is designed to accommodate a specific format of address and reference data. The geocoding style template file is distinguished with an .lot file extension.

geodatabase

An object-oriented geographic database that provides services for managing geographic data. These services include validation rules, relationships, and topological associations. A geodatabase contains feature datasets and is hosted inside a relational database management system.

geographic base files (GBF)

See reference files.

GEOPUB30.DLL

An ArcView Geocoding Windows dynamic link library (DLL) for use on ArcView 3.x. It exposes additional geocoding requests by making them public (rather than private), Avenue scripts-callable requests.

GETMAXSCORE

Calculates the maximum composite score based on a .mat file. You only need this tool for ArcView 3.x.

input event records

Input event record types vary by application. They include customer addresses, location of the event or incident, location of equipment and facilities, and the monument offset.

intersection

The crossing point of two line features. In geocoding, this most often refers to the crossing point of two streets.

intersection connector

A character used in the address data to indicate that the address consists of two line features. For example, in the address S. Huntington Dr. & E. Clark Blvd., the ampersand (&) character is the intersection connector. These characters are used to assign intersection searches to specific files in the rule base.

keyword

A single word that appears in the address. A standard abbreviation is substituted for the keyword in the .cls file.

literals

Character constants that are represented by enclosing a string in quotes.

location

A geographic identification assigned to a region or feature based on a specific coordinate system. In the geocoding process, the location is defined with an x,y coordinate value according to the distance north or south of the equator and east or west of the prime meridian.

m probability

The probability that the field agrees given the record pair is a match. The m probability is one minus the error rate of the field in a matched record. The more reliable a field is, the greater the m probability will be.

MATCH commands

Used to specify the comparison type, match key field, variable name, matching probabilities, and additional parameters when defining match rules for match variables.

match file

A file in the geocoding rule base that defines variables for the address items found in the reference file attribute table. The filename is always <file>.mat.

matching

Performing detailed candidate scoring and field-by-field comparisons to find a composite score for each candidate.

match key

A coded element consisting of two characters, used to define and process specific address elements in the geocoding rule base.

match key contents

The match key field name enclosed in braces.

match key dictionary

A file in the geocoding rule base that defines the data type, field length, and missing value code for each match key field. The file name is always <file>.dct.

match key fields

Fields defined in the match key dictionary that contain the data type, field length, and missing value code.

MatchRules

Define variables for the address items found in the reference file attribute table.

match score

A value assigned to all potential candidates of an address match. The match score is based on how well the location found in the reference data matches with the address data being searched.

minimum match score

The minimum score a match candidate needs to be considered in the geocoding search. This value is adjustable in the Geocoding Service Properties dialog box.

MOVE

A command in the pattern file that moves a user variable or a match key field to another match key field. When you move information using this command, the source is made null after it is moved.

negation class qualifier

A qualifier in the pattern file denoted by !. This qualifier indicates NOT and will match to any token except a street type. It can be combined with a floating class only at the beginning of a pattern.

normalized

An internal process (except for ArcView 3.x) where the computed maximum score is recalculated to be between 0 and 100 based on a formula. Normalizing the score allows you to see how good a match is compared to other matches.

operands

A parameter that is referred in the program. Programming routines or actions can be executed to the specific operand.

OUTFILE

Keyword in the command file that sets the debugging output to <file_name>.

parsing parameters

Default rules in the command file that define what constitutes a token or operand as defined in the pattern file.

pattern–action sequence

Pairs of patterns and actions in the pattern file that work in conjunction with each other. A pattern file can contain as many pattern–action sequences as you want.

pattern classes

Elements in an address are classified using pattern classes in the pattern file. Some examples of pattern classes include the numeric class, single alphabetic word, and unknown. The .cls file also supplies classes A–Z.

pattern file

A file in the geocoding rule base that contains pattern rules and actions for standardizing an address and converting the recognized operands into match key fields. The file name is always <file>.pat.

pattern rules

Rules in the pattern file. There are two types of pattern rules: conditional and unconditional. See also conditional patterns and unconditional patterns.

place name alias

The formal or common name of a location, such as the name of a school, hospital, or other landmark. In geocoding, the geocoding service can be designed to accommodate the use of a place name alias as a search parameter.

PREFIX

A command in the pattern file that adds the concatenated operand to the beginning of the string.

PREFIX_A

A command in the pattern file that prefixes the standard abbreviation instead of the raw data. The source must be an operand.

prefix values

Values in an address that are assigned as prefixes by the .cls file. For example, in the address 3453 West Santa Monica Blvd., West is the prefix value.

primary reference data

The most basic of reference material used in a geocoding service, usually consisting of the geometry of a region and its associated attribute table.

primary table

The attribute table associated with the primary reference data. Based on the address locator style selected, certain address elements must be present in the primary table.

qualifier

A symbol in the pattern file that qualifies the token it is matched with. For example, the ! symbol qualifies the token to be NOT. If the ! is matched to D, and the direction is north, then the qualifier indicates NOT north.

reference data

The material containing location information and address information of specific features. Reference data consists of the geospatial representation of the data, the related attribute table, and the geocoding indexes. The place name alias table and the alternate name table are also considered reference data.

All material contained in the Address data model used as the reference material in a specific geocoding service.

reference files

Along with input event records and software, reference files (which are often referred to as Geographic Base Files) are part of the geocoding process. They can vary from simple digitized boundary files to more complex address coding guides to even more sophisticated centerline files.

region

A specified, usually large, and continuous area of a geographic surface.

rematch

The process of reassigning the geocoding results to other locations.

RETURN

A command in the pattern file that returns control from a subroutine to the main program.

RETYPE

A command in the pattern file that lets you change the type of an operand in the token table, change the value of an operand, or change the abbreviation of an operand if it is found in the classification table. You can also retype an operand to a null class, or change the token type and replace the text of a token.

reverse floating position specifier

A specifier in the pattern file denoted by #. This specifier is similar to the floating position specifier, except that scanning proceeds from left to right instead of right to left. See also floating position specifier.

rule base

A collection of files that directs the geocoding engine in how to standardize address data and match it to the related location in the reference data. Each geocoding style uses specific files in the rule base.

SDE®

See Spatial Database Engine™.

secondary reference data

All other material used in a geocoding service as reference data beyond the primary reference data. This can consist of an alternate name table or a place name alias table. This entity becomes null when using an address data model as reference material.

SEPLIST

A list of characters or symbols used to define separate tokens. See also parsing parameters.

series of conditional values

Specified by delimiting the conditional values using either spaces or commas.

server-side address locator

An address locator where processing is done on one computer with the results accessible to other computers. Server-side address locators consist of services available over the Internet, via local area networks, or via an ArcSDE server.

shapefile

A vector data storage format for storing the location, shape, and attributes of geographic features. A shapefile is stored in a set of related files and contains one feature class.

side offset

A distance off of the street feature. The side offset is an adjustable value that dictates how far away from the line feature an address location should be matched. Using a side offset will eliminate the point feature from being placed directly over the line feature.

simple conditional values

Denoted by the equality operator and a value. The equality operator tests both the standardized abbreviation and the original token value for equality to the operand.

single token

A pattern class in the pattern file defined by an ampersand (&).

Soundex

An index search key that finds the match plus all the potential candidates.

Spatial Database Engine™ (SDE®)

A gateway to a multiuser commercial relational database management system (RDBMS)—for example, Oracle®, Microsoft SQL Server™, Informix®, or DB2®. ArcSDE is an open, high-performance spatial data server that employs client/server architecture to perform efficient spatial operations and manage large, shared geographic data. It has been referred to as ArcSDE since 1999.

specifier

A symbol that specifies what action should be taken with an address token in the pattern file.

spelling sensitivity

The degree to which a geocoding service must match the spelling of one address element in the address data to the address element in the reference data.

standardization

1. A process of address parsing that prepares the address to be matched against the reference data.
2. A two-step process to standardize data. The first step involves preparing the reference file, and the second step involves preparing the events for matching.

standardization process

Contains standardization commands (<file>.stn), a match key dictionary for the event address (<file>.dct), a classification table for providing standard address abbreviations (<file>.cls), and patterns and actions for standardizing an address (<file>.pat). The standardization process can also include additional optional tables.

STANEDIT

The Windows version of Interactive Standardizer. It is used for pattern rules syntax checking and debugging.

STRIPLIST

A list of characters or symbols to be stripped during the standardization. See also parsing parameters.

subfield classes

Classes in the pattern file used to pick off individual words or a ? string. They range from 1 to 9 and -1 to -9. They are useful for processing address suffixes, such as 123-A Main St.

subfield ranges

A class in the pattern file used to specify a range of unknown words. Subfield ranges are specified as (beg:end).

subroutines

Actions in the pattern file that are called to perform a particular function (such as CALL_DIRECTIONS, where DIRECTIONS is the subroutine).

table of conditional values

An ASCII file with one line for each conditional value. Tables are specified by @<table file name>.

token

A delimited word, string, or symbol from the entire input string.

token type value

A value assigned to a particular value in the address by the .cls file.

tokenizing

The process of assigning a token to a part of an address in order to code it.

u probability

The probability that the field agrees given the record pair is unmatched (that is, the probability that the field agrees at random).

unconditional patterns

One type of pattern rule. Unconditional pattern rules are strict rules that are insensitive to the value of the operand.

universal class

A class in the pattern file denoted by **. The universal class matches all tokens. It can be combined with other operands to restrict the tokens grabbed by the class.

user interface

The method of interaction between the user and the software. In the geocoding framework, the user interface consists of ArcCatalog, ArcMap, and ArcToolbox™ as well as any associated dialog boxes.

VAR commands

Commands in the match file used to specify variable names, field position, field length, and missing value codes.

variables

Can be either numeric or character type. If numeric, its type is set to a numeric value. If character, its type is set to a literal value.

VARTYPE commands

The last line of the match file after you have defined the MatchRules. It is internal, but VARTYPE indicates that frequency analyses won't be performed.

Web service

A software component accessible over the World Wide Web for use in other applications. Web services are built using industry standards such as XML and SOAP and, thus, are not dependent on any particular operating system or programming language, allowing access through a wide range of applications.

ZIP

1. Zone Improvement Plan. ZIP Code identifies a specific geographic delivery area.
2. The postal code system used by the United States Postal Service.

zone

Additional information about a location or address, used to narrow the geocoding search and increase search speed. Address elements and their related locations such as city, postal code, or country all can act as a zone.

Index

A

Actions
defined 151. *See also* Pattern file: actions: defined
described 55–59

Adding files to folder
checklist 74

Address data
defined 151
described 3

Address data format
defined 151

Address data model
defined 151

Address element
defined 151

Address field
defined 151

Address format
defined 151

Address locator
defined 152

Address Locator Properties dialog box
defined 152

Address matching
defined 152
described 2

Address standardizer
defined 152

Address styles
available 35

Addresses
defined 151
entering multiple 71

Alternate name
defined 152

API
defined 152

Application programming interface
defined 152

Arithmetic expressions. *See also* Pattern rules:
conditional expressions: arithmetic
expressions
defined 152
described 54

B

Blocking. *See also* Geocoding process: blocking
defined 152
described 4, 5

Booster index
defined 152

C

Candidate
defined 152
scores 20

Class
defined 152
simple pattern classes 48

Classification table 10
comparison threshold
defined 153
described 36
defined 152
modifying 40
opening 37
overview 36
set up 38

Classifying
defined 153
described 11

Client-side address locator
defined 153

Command file
defined 153

Comparison threshold
defined 153
described 36

- Composite weight
 - defined 153
 - described 20
- CONCAT
 - defined 153. *See also* Pattern file: actions: concatenating information
 - described 56
- CONCAT_A
 - defined 153. *See also* Pattern file: actions: concatenating information
- Conditional expressions
 - combining 54
 - defined 153
 - described 52, 54
 - relational operators 52
- Conditional pattern values
 - defined 153
 - described 51–52
- Conditional patterns. *See also* Pattern rules: conditional patterns
 - defined 153
 - described 51
- Constants
 - defined 153
 - referencing 53
- CONVERT
 - defined 153. *See also* Pattern file: actions: CONVERT
 - described 57
- COPY
 - defined 153. *See also* Pattern file: actions: COPY commands
- COPY commands
 - described 55–56
- COPY_A
 - defined 153. *See also* Pattern file: actions: COPY commands
- COPY_S
 - defined 154. *See also* Pattern file: actions: COPY commands

D

- DEBUG
 - adding keyword to command file 27, 70
 - defined 154
- Dynamic feature class
 - defined 154

E

- ENCODPAT 73
 - defined 154
 - encoding .xat files 73
 - running from DOS 73
- End of field
 - defined 154
 - described 49
- End offset
 - defined 154
- EXIT
 - defined 154
 - described 58

F

- Fields
 - adding to match key dictionary 32, 33
 - removing from match key dictionary 33
- File
 - defined 154
- Fixed position specifier
 - defined 154
 - described 50
- Floating position specifier
 - defined 154
 - described 49–50

G

- GBFs (Geographic Base Files)
 - defined 154
 - described 3
- GDB
 - defined 154
- Geocode
 - defined 154
 - described 2
 - general process 4–7
- Geocoding Editor
 - defined 154
- Geocoding engine
 - defined 155
- Geocoding index
 - defined 155
- Geocoding platform
 - defined 155
- Geocoding process 4–7
 - address data 3
 - blocking 5–6
 - components 3
 - defined 155
 - defining matching strategies 4
 - matching 6
 - reference data 3
 - review/edit 7
 - software 3
 - standardization 4–5
- Geocoding software
 - described 3
- Geocoding style
 - defined 155
- Geodatabase
 - defined 155
- Geographic base files (GBFs)
 - defined 155
 - described 3

GEOPUB30.DLL
defined 155
GETMAXSCORE
defined 155

I

Input event records
defined 155
Intersection connector
defined 155
Intersections
dealing with 64–65
defined 155
defining MatchRules 64
editing .xat/.pat files 66
matching and standardizing 64
writing pattern rules 64–65

K

Keyword
defined 156
described 36

L

Left/right interval comparisons
described 6
Literals
defined 156
referencing 53
Location
defined 156

M

m probability 12, 19
defined 156
described 12
MATCH commands 16–17
defined 156
format 16–17
Match file
defined 156
MATCH commands 16–17
format 16–17
MatchRules 14–15
modifying 21
overview 14–15
VAR commands 15
format 15
VARTYPE commands 18
format 18
Match key
contents
defined 156
referencing 53
defined 156
described 4
dictionary 10
defined 156
modifying 32
fields
defined 156
Match key dictionary
dictionary
overview 30–31
Match score
defined 156
Matching
defined 156

Matching weights 20
MatchRules 14–15
defined 156
described 12
m probability 12
MATCH commands
m probability 19
u probability 19
u probability 12
Minimum match score
defined 156
MOVE
defined 156
described 56

N

Negation class qualifier
defined 157
described 50
Normalized
defined 157

O

Operands
defined 157
referencing current contents 52–53
referencing length of 53
referencing substring 54
referencing template 53
OUTFILE
adding keyword to command file 27, 70
defined 157

P

Parsing parameters 25
adjusting 26
defined 157

- Parsing parameters (continued)
 - SEPLIST 25
 - defined 159
 - STRIPLIST 25
 - defined 160
- Pattern classes
 - defined 157
- Pattern file 10
 - actions 55–59. *See also* Actions
 - concatenating information 56
 - CONVERT 57
 - COPY commands 55–56
 - defined 151
 - EXIT 58
 - MOVE 56
 - RETYPE 57
 - reverse Soundex 58
 - Soundex 58
 - subroutine 58–59
 - adding custom routines 67–68
 - adding quickly filterable patterns 61
 - creating new process 68
 - by editing existing process 68
 - dealing with ST 63
 - defined 157
 - encoding 60
 - format 46
 - modifying 60
 - modifying for a special circumstance 62
 - overview 46–47
- Pattern rules 48–54
 - conditional expressions 52
 - arithmetic expressions 54. *See also* Arithmetic expressions
 - combining conditional expressions 54
 - referencing a substring of an operand 54
 - referencing a template of an operand 53
 - referencing constants 53
 - referencing current operand contents 52–53
 - referencing literals 53
 - referencing match key contents 53

- Pattern rules (continued)
 - referencing the length of an operand 53
 - referencing variables 53
- conditional patterns 51–54
 - series of conditional values 51
 - simple conditional values 51
 - tables of conditional values 51–52
- defined 157
- unconditional patterns 48–50
 - end of field 49
 - fixed position specifier 50
 - floating position specifier 49–50
 - negation class qualifier 50
 - reverse floating position specifier 50
 - single special character classes 48
 - single token 49
 - subfield classes 49
 - subfield ranges 49
 - universal class 49
- Pattern-action sequence
 - defined 157
- Place name alias
 - defined 157
- PREFIX
 - defined 157
- Prefix values
 - defined 157
- PREFIX_A
 - defined 157
- Primary reference data
 - defined 157
- Primary table
 - defined 158
- Probabilistic record linkage systems 3

Q

- Qualifier
 - defined 158

R

- Reference data
 - defined 158
 - described 3
- Reference files
 - defined 158
- Region
 - defined 158
- Rematch
 - defined 158
- RETURN
 - defined 158
 - described 59
- RETYPE
 - defined 158
 - described 57
- Reverse floating position specifier
 - defined 158
 - described 50
- Reverse Soundex. *See also* Pattern file: actions:
 - reverse Soundex
 - described 58
- Rule base
 - defined 158

S

- SDE
 - defined 158
- Secondary reference data
 - defined 158

SEPLIST. *See* Parsing parameters: SEPLIST
Series of conditional values
 defined 159
 described 51
Server-side address locator
 defined 159
Shapefile
 defined 159
Side offset
 defined 159
Simple conditional values
 defined 159
 described 51
Single token
 defined 159
 described 49
Soundex 5–6. *See also* Pattern file: actions:
 Soundex
 defined 159
 described 58
Spatial Database Engine
 defined 159
Specifier
 defined 159
Spelling sensitivity
 defined 159
Standardization
 commands 10
 defined 159
 process 10
 classification table 10
 defined 159
 match key dictionary 10
 pattern file 10
 standardization commands 10
STANEDIT 70
 checking pattern rules syntax 71
 creating new process from 72
 defined 160
 standardizing addresses 71

Street intersections. *See* Intersections
STRIPLIST. *See* Parsing parameters:
 STRIPLIST
Subfield classes
 defined 160
 described 49
Subfield ranges
 defined 160
 described 49
Subroutines
 defined 160
 described 58

T

Tables of conditional values
 defined 160
 described 51–52
Token
 defined 160
Token type value
 defined 160
Tokenizing
 defined 160
 described 11

U

u probability 12, 19
 defined 160
 described 12
Unconditional patterns. *See also* Pattern rules:
 unconditional patterns
 defined 160
Universal class
 defined 160
 described 49
User interface
 defined 160

V

VAR commands 15
 defined 160
 format 15
Variables
 defined 160
 referencing 53
VARTYPE commands 18
 defined 160
 format 18

W

Web service
 defined 161

Z

ZIP
 defined 161
Zone
 defined 161

